GNU Make 使用手冊

Version 3.79

繁體版更新日期 2006/08/30
校稿　ChanningLan

目錄

1 Make 概述

Make 可自動決定一個大程式中哪些檔案需要重新編譯，並發布重新編譯它們的命令。本版本GNU Make使用手冊由Richard M. Stallman and Roland McGrath編著，是從Paul D. Smith撰寫的V3.76版本發展過來的。

GNU Make符合IEEE Standard 1003.2-1992 (POSIX.2) 6.2章節的規定。

因爲C語言程式更具有代表性，所以我們的例子基於C語言程式，但Make並不是僅僅能夠處理C語言程式，它可以處理那些編譯器能夠在Shell命令下執行的的各種語言的程式。事實上，GNU Make不僅僅限於程式，它可以適用於任何如果一些檔案變化導致另外一些檔案必須更新的任務。

如果要使用Make，必須先寫一個稱爲Makefile的檔案，該檔案描述程式中各個檔案之間的相互關係，並且提供每一個檔案的更新命令。在一個程式中，可執行程式檔案的更新依靠OBJ檔案，而OBJ檔案是由源檔案編譯得來的。

一旦合適的Makefile檔案存在，每次更改一些源檔案，在shell命令下簡單的鍵入：

make

就能執行所有的必要的重新編譯任務。Make程式根據Makefile檔案中的數據和每個檔案更改的時間戳決定哪些檔案需要更新。對於這些需要更新的檔案，Make基於Makefile檔案發布命令進行更新，進行更新的模式由提供的命令行參數控制。具體操作請看執行Make章節。

1.1怎樣閱讀本手冊

如果您現下對Make一無所知或者您僅需要了解對make 的普通性介紹，請查閱前幾章內容，略過後面的章節。前幾章節是普通介紹性內容，後面的章節是具體的專業、技術內容。

如果您對其它Make程式十分熟悉，請參閱GNU Make的特點和不相容性和失去的特點部分，GNU Make的特點這一章列出了GNU Make對make程式的展開，不相容和失去的特點一章解釋了其它Make程式有的特徵而GNU Make缺乏的原因。

對於快速瀏覽者，請參閱選項概要、快速參考和內建的特殊目標名部分。

1.2問題和BUG

如果您有關於GNU Make的問題或者您認爲您發現了一個BUG，請向開發者報告；我們不能許諾我們能幹什麼，但我們會盡力修正它。在報告BUG之前，請確定您是否真正發現了BUG，仔細研究文檔後確認它是否真的按您的指令執行。如果文檔不能清楚的告訴您怎么做，也要報告它，這是文檔的一個BUG。

在您報告或者自己親自修正BUG之前，請把它分離出來，即在使問題暴露的前提下儘可能的縮小Makefile檔案。然後把這個Makefile檔案和Make給出的精確結果發給我們。同時請說明您希望得到什麼，這可以幫助我們確定問題是否出在文檔上。

一旦您找到一個精確的問題，請給我們發E-mail，我們的E-mail位址是：

bug-make@gnu.org

在郵件中請引入(include)您使用的GNU Make的版本號。您可以利用命令‘make—version’得到版本號。同時希望您提供您的機器型號和作業系統類型，如有可能的話，希望同時提供config.h檔案（該檔案有配置過程產生）。

2 Makefile檔案介紹

Make程式需要一個所謂的Makefile檔案來告訴它干什麼。在大多數情況下，Makefile檔案告訴Make怎樣編譯和連接成一個程式。

本章我們將討論一個簡單的Makefile檔案，該檔案描述怎樣將8個C源程式檔案和3個頭檔案編譯和連接成為一個文字(text)編輯器。Makefile檔案可以同時告訴Make怎樣執行所需要的雜亂無章的命令（例如，清除操作時刪除特定的檔案）。如果要看更詳細、複雜的Makefile檔案例子，請參閱複雜的Makefile檔案例子一章。

當make重新編譯這個編輯器時，所有改動的C語言源檔案必須重新編譯。如果一個頭檔案改變，每一個引入(include)該頭檔案的C語言源檔案必須重新編譯，這樣才能保證生成的編輯器是所有源檔案更新後的編輯器。每一個C語言源檔案編譯後產生一個對應的OBJ檔案，如果一個源檔案重新編譯，所有的OBJ檔案無論是剛剛編譯得到的或原來編譯得到的必須從新連接，形成一個新的可執行檔案。

2.1 規則的格式

一個簡單的Makefile檔案引入(include)一系列的“規則”，其樣式如下：

目標(target)…: 先決條件(prerequiries)…
<tab>命令(command)
…
…

目標(target)通常是要產生的檔案的名稱，例如;目標可以是執行檔案或OBJ檔案。目標也可是一個執行的動作名稱，諸如‘clean’（詳細內容請參閱假想(phony)目標一節）。

先決條件是用來輸入從而產生目標的檔案，一個目標經常有幾個先決條件。
命令是Make執行的動作，一個規則可以含有幾個命令，每個命令占一行。**注意：每個命令行前面必須是一個Tab字符，即命令行第一個字符是Tab**。這是不小心容易出錯的地方。

通常，如果一個先決條件發生變化，則需要規則呼叫命令對相應先決條件和服務進行處理從而更新或建立目標。但是，指定命令更新目標的規則並不都需要先決條件，例如，引入(include)和目標‘clern’相聯繫的刪除命令的規則就沒有先決條件。

規則一般是用於解釋怎樣和何時重建特定檔案的，這些特定檔案是這個詳盡規則的目標。Make需首先呼叫命令對先決條件進行處理，進而才能建立或更新目標。當然，一個規則也可以是用於解釋怎樣和何時執行一個動作，詳見編寫規則一章。

一個Makefile檔案可以引入(include)規則以外的其它文字(text)，但一個簡單的Makefile檔案僅僅需要引入(include)規則。雖然真正的規則比這裡展示的例子複雜，但格式卻是完全一樣。

2.2一個簡單的Makefile檔案

一個簡單的Makefile檔案，該檔案描述了一個稱為文字(text)編輯器（edit）的可執行檔案生成方法，該檔案依靠8個OBJ檔案（.o檔案），它們又依靠8個C源程式檔案和3個頭檔案。
在這個例子中，所有的C語言源檔案都引入(include)‘defs.h’頭檔案，但僅僅定義編輯命令的源檔案引入(include)‘command.h’頭檔案，僅僅改變編輯器緩沖區的低層檔案引入(include)‘buffer.h’頭檔案。

```
edit : main.o kbd.o command.o display.o \
     insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
```

```
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

我們把每一個長行使用反斜線(\)分裂爲兩行或多行，實際上它們相當於一行，這樣做的意圖僅僅是爲了閱讀方便。

使用Makefile檔案建立可執行的稱爲‘edit’的檔案，鍵入：make

使用Makefile檔案從目錄中刪除可執行檔案和目標，鍵入：make clean

在這個Makefile檔案例子中，目標包括可執行檔案‘edit’和OBJ檔案‘main.o’及‘kdb.o’。先決條件是C語言源檔案和C語言頭檔案如‘main.c’和‘def.h’等。事實上，每一個OBJ檔案即是目標也是先決條件。所以命令行包括‘cc -c main.c’和‘cc -c kbd.c’。

當目標是一個檔案時，如果它的任一個先決條件發生變化，目標必須重新編譯和連接。任何命令行的第一個字符必須是‘Tab’字符，這樣可以把Makefile檔案中的命令行與其它行分別開來。（一定要牢記：Make並不知道命令是如何工作的，它僅僅能向您提供保證目標的合適更新的命令。**Make的全部工作是當目標需要更新時，按照您製定的具體規則執行命令。）**

目標‘clean’不是一個檔案，僅僅是一個動作的名稱。正常情況下，在規則中‘clean’這個動作並不執行，目標‘clean’也不需要任何先決條件。一般情況下，除非特意告訴make執行‘clean’命令，否則‘clean’命令永遠不會執行。注意這樣的規則不需要任何先決條件，它們存在的目的僅僅是執行一些特殊的命令。像這些不需要先決條件僅僅表達動作的目標稱爲假想(phony)目標。詳細內容參見假想(phony)目標；參閱命令錯誤可以了解rm或其它命令是怎樣導致make忽略錯誤的。

2. make處理makefile檔案的過程

預設情況下，make開始於第一個目標（假想(phony)目標的名稱前帶‘.’）。這個目標稱爲預設最終目標（即make最終更新的目標，具體內容請看指定最終目標的參數一節）。

在上節的簡單例子中，預設最終目標是更新可執行檔案‘edit’，所以我們將該規則設爲第一規則。這樣，一旦您給出命令：

make

make就會讀當前目錄下的makefile檔案，並開始處理第一條規則。在本例中，第一條規則是連接生成‘edit’，但在make全部完成本規則工作之前，必須先處理‘edit’所依靠的OBJ檔案。這些OBJ檔案按照各自的規則被處理更新，每個OBJ檔案的更新規則是編譯其源檔案。重新編譯根據其依靠的源檔案或頭檔案是否比現存的OBJ檔案更‘新’，或者OBJ檔案是否存在來判斷。

其它規則的處理根據它們的目標是否和預設最終目標的先決條件相關聯來判斷。如果一些規則和預設最終目標無任何關聯則這些規則不會被執行，除非告訴Make強製執行（如輸入執行make clean命令）。

在OBJ檔案重新編譯之前，Make首先檢查它的先決條件C語言源檔案和C語言頭檔案是否需要更新。如果這些C語言源檔案和C語言頭檔案不是任何規則的目標，make將不會對它們做任何事情。Make也可以自動產生C語言源程式，這需要特定的規則，如可以根據Bison或Yacc產生C語言源程式。

在OBJ檔案重新編譯（如果需要的話）之後，make決定是否重新連接生成edit可執行檔案。如果edit可執行檔案不存在或任何一個OBJ檔案比存在的edit可執行檔案‘新’，則make重新連接生成edit可執行檔案。

這樣，如果我們修改了‘insert.c’檔案，然後執行make，make將會編譯‘insert.c’檔案更新‘insert.o’檔案，然後重新連接生成edit可執行檔案。如果我們修改了‘command.h’檔案，然後執行make，make將會重新編譯‘kbd.o’和‘command.o’檔案，然後重新連接生成edit可執行檔案。

2.4使用變數簡化makefile檔案

在我們的例子中，我們在‘edit’的生成規則中把所有的OBJ檔案列舉了兩次，這裡再重複一遍：

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o
```

這樣的兩次列舉有出錯的可能，例如在系統中加入一個新的OBJ檔案，我們很有可能在一個需要列舉的地方加入了，而在另外一個地方卻忘記了。我們使用變數可以簡化makefile檔案並且排除這種出錯的可能。變數是定義一個字元串一次，而能在多處替代該字元串使用（具體內容請閱讀使用變數一節）。

在makefile檔案中使用名爲objects, OBJECTS, objs, OBJS, obj, 或 OBJ的變數代表所有OBJ檔案已是約定成俗。在這個makefile檔案我們定義了名爲objects的變數，其定義格式如下：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

然後，在每一個需要列舉OBJ檔案的地方，我們使用寫爲`$(objects)'形式的變數代替（具體內容請閱讀使用變數一節）。

下面是使用變數後的完整的makefile檔案：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

2.5 讓make推斷命令

編譯單獨的C語言源程式並不需要寫出命令，因爲make可以把它推斷出來：make有一個使用‘CC　c’命令的把C語言源程式編譯更新爲相同檔案名的OBJ檔案的隱含規則。例如make可以自動使用‘cc -c main.c -o main.o’命令把‘main.c’編譯‘main.o’。因此，我們可以省略OBJ檔案的更新規則。詳細內容請看使用隱含規則一節。

如果C語言源程式能夠這樣自動編譯，則它同樣能夠自動加入到先決條件中。所以我們可在先決條件中省略C語言源程式，進而可以省略命令。下面是使用隱含規則和變數objects的完整makefile檔案的例子：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

這是我們實際編寫makefile檔案的例子。（和目標‘clean’聯繫的複雜情況在別處闡述。具體參見假想(phony)目標及命令錯誤兩節內容。）因爲隱含規則十分方便，所以它們非常重要，在makefile檔案中經常使用它們。

2.6 另一種風格的makefile檔案

當時在makefile檔案中使用隱含規則建立OBJ檔案時，採用另一種風格的makefile檔案也是可行的。在這種風格的makefile檔案中，可以依據先決條件分組代替依據目標分組。下面是採用這種風格的makefile檔案：

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
edit : $(objects)
        cc -o edit $(objects)
$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

這裡的defs.h是所有OBJ檔案的共同的一個先決條件；command.h和bufffer.h是具體列出的OBJ檔案的共同先決條件。

雖然採用這種風格編寫makefile檔案更具風味：makefile檔案更加短小，但一部分人以爲把每一個目標的訊息放到一起更清晰易懂而不喜歡這種風格。

2.7 在目錄中刪除檔案的規則

編譯程式並不是編寫make規則的唯一事情。Makefile檔案可以告訴make去完成編譯程式以外的其它任務，例如，怎樣刪除OBJ檔案和可執行檔案以保持目錄的‘乾淨’等。下面是刪除利用make規則編輯器的例子：

```
clean:
        rm edit $(objects)
```

在實際應用中，應該編寫較爲複雜的規則以防不能預料的情況發生。更接近實用的規則樣式如下：

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

這樣可以防止make因爲存在名爲’clean’的檔案而發生混亂，並且導致它在執行rm命令時發生錯誤（具體參見假想(phony)目標及命令錯誤兩節內容）。

諸如這樣的規則不能放在makefile檔案的開始，因爲我們不希望它變爲預設最終目標。應該像我們的makefile檔案例子一樣，把守關口於edit的規則放在前面，從而把編譯更新edit可執行程式定爲預設最終目標。

3 編寫makefile檔案

make編譯系統依據的訊息來源於稱為makefile檔案的資料庫。

3.1 makefile檔案的內容

- **makefile檔案包含5種內容：具體規則、隱含規則、定義變數、指令和註釋。規則、變數和指令**將在後續章節介紹。
- **具體規則**用於闡述什么時間或怎樣重新生成稱為規則目標的一個或多個文件的。它列舉了目標所依靠的檔案，這些檔案稱為該目標的先決條件。具體規則可能同時提供了建立或更新該目標的命令。詳細內容參閱編寫規則一章。
- 隱含規則用於闡述什么時間或怎樣重新生成同一文件名的一系列文件的。它描述的目標是根據和它名字相同的檔案進行建立或更新的，同時提供了建立或更新該目標的命令。詳細內容參閱使用隱含規則一節。
- 定義變數是為一個變數賦一個固定的字符串值，從而在以後的文件中能夠使用該變數代替這個字符串。注意在makefile檔案中定義變數占一獨立行。在上一章的makefile檔案例子中我們定義了代表所有OBJ檔案的變數objects（詳細內容參閱使用變數簡化makefile檔案一節）。
  - 指令是make根據makefile文件執行一定任務的命令。這些包括如下幾方面:
  - 讀取其它makefile文件（詳細內容參見*引入(include)其它的makefile文件*）。
  - 判定（根據變數的值）是否使用或忽略makefile文件的部分內容（詳細內容參閱***makefile文件的條件語句***一節）。
  - 定義多行變數，即定義變數值可以引入(include)多行字符的變數（詳細內容參見***定義多行變數***一節）。

以‘#’開始的行是注釋行。註釋行在處理時將被make忽略，如果一個註釋行在行尾是‘\’則表示下一行繼續為註釋行，這樣註釋可以持續多行。除在define指令內部外，註釋可以出現下makefile檔案的任何地方，甚至在命令內部（這裡shell決定什麼是註釋內容）。

3.2 makfile檔案的命名

預設情況下，當make尋找makefile檔案時，它試圖搜尋具有如下的名字的檔案，按順序：‘GNUmakefile’、‘makefile’和‘Makefile’。

通常情況下您應該把您的makefile檔案命名為‘makefile’或‘Makefile’。(我們推薦使用‘Makefile’，因為它基本出現下目錄清單的前面，後面挨著其它重要的檔案如‘README’等。)。雖然首先搜尋‘GNUmakefile’，但我們並不推薦使用。除非您的makefile檔案是特為GNU make編寫的，在其它make版本上不能執行，您才應該使用‘GNUmakefile’作為您的makefile的檔案名。

如果make不能發現具有上面所述名字的檔案，它將不使用任何makefile檔案。這樣您必須使用命令參數給定目標，make試圖利用內建的隱含規則確定如何重建目標。詳細內容參見使用隱含規則一節。

如果您使用非標準名字makefile檔案，您可以使用‘-f’或‘--file’參數指定您的makefile檔案。參數‘-f name’或‘--file=name’能夠告訴make讀名字為‘name’的檔案作為makefile檔案。如果您使用‘-f’或‘--file’參數多於一個，意味著您指定了多個makefile檔案，所有的makefile檔案按具體的順序發生作用。一旦您使用了‘-f’或‘--file’參數，將不再自動檢查是否存在名為‘GNUmakefile’、‘makefile’或‘Makefile’的makefile檔案。

3.3 引入(include)其它的makefile檔案

include指令告訴make暫停讀取當前的makefile檔案，先讀完include指令指定的makefile檔案後再繼續。指令在makefile檔案占單獨一行，其格式如下：

```
include filenames...
```

filenames可以包含shell檔案名的格式。

在include指令行，行開始處的多餘的空格是允許的，但make處理時忽略這些空格，**注意該行不能以Tab字符開始（因為，以Tab字符開始的行，make認為是命令行）**。include和檔案名之間以空格隔開，兩個檔案名之間也以空格隔開，多餘的空格make處理時忽略，在該行的尾部可以加上以‘#’為起始的註釋。檔案名可以引入(include)變數及函數呼叫，它們在處理時由make進行展開（具體內容參閱使用變數一節）。

例如，有三個‘.mk’檔案：‘a.mk’、‘b.mk’和‘c.mk’，變數$(bar)展開爲bish bash，則下面的表達是：

include foo *.mk $(bar)
和‘include foo a.mk b.mk c.mk bish bash’相同。

當make遇見include指令時， make就暫停讀取當前的makefile檔案，依次讀取列舉的makefile檔案，讀完之後，make再繼續讀取當前makefile檔案中include指令以後的內容。
使用include指令的一種情況是幾個程式分別有單獨的makefile檔案，但它們需要一系列共同的變數定義（詳細內容參閱設定變數），或者一系列共同的樣式規則（詳細內容參閱定義與重新定義樣式規則）。

另一種使用include指令情況是需要自動從源檔案爲目標產生先決條件的情況，此時，先決條件在主makefile檔案引入(include)的檔案中。這種模式比其它版本的make把先決條件附加在主makefile檔案後部的傳統模式更顯得簡潔。具體內容參閱自動產生先決條件。
如果makefile檔案名不以‘/’開頭，並且在當前目錄下也不能找到，則需搜尋另外的目錄。首先，搜尋以‘-I’或‘--include-dir’參數指定的目錄，然後依次搜尋下面的目錄（如果它們存在的話）：‘prefix/include' (通常爲‘/usr/local/include') ‘/usr/gnu/include', ‘/usr/local/include', ‘/usr/include'。

如果指定引入(include)的makefile檔案在上述所有的目錄都不能找到，make將產生一個警告訊息，注意這不是致命的錯誤。處理完include指令引入(include)的makefile檔案之後，繼續處理當前的makefile檔案。一旦完成makefile檔案的讀取操作，make將試圖建立或更新舊式的或不存在的makefile檔案。詳細內容參閱makefile檔案重新生成的過程。只有在所有make尋求丟失的makefile檔案的努力失敗後，make才能斷定丟失的makefile檔案是一個致命的錯誤。
如果您希望對不存在且不能重新建立的makefile檔案進行忽略，並且不產生錯誤訊息，則使用-include指令代替include指令，格式如下：

-include filenames...

這種指令的作用就是對於任何不存在的makefile檔案都不會產生錯誤（即使警告訊息也不會產生）。如果希望保持和其它版本的make兼容，使用sinclude指令代替-include指令。

3.4 變數MAKEFILES

當 make 讀取數個 makefile 時（包括根據環境變數 MAKEFILES 讀取的 makefile，命令列指定的、預設的、使用 include 指定的），這些makefile的名字會被自動的加進變數 MAKEFILE_LIST 之中。這些檔案名稱會在開始被 make 解析之前就被加進變數 MAKEFILE_LIST 。
也就是說，如果在一個 makefile 中做的第一件事就是檢查這個變數的最後一個字，這個字就該是目前這個 makefile 的檔名。一旦目前的 makefile 使用了指令 include ，這個變數的最後一個字就會變成被 include 進來的 makefile 的檔名。
若一個名爲 Makefile 的 makefile 含有以下內容：

```
name1 := $(word $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST))
include inc.mk
name2 := $(word $(words $(MAKEFILE_LIST)),$(MAKEFILE_LIST))

all:
@echo name1 = $(name1)
@echo name2 = $(name2)
```

那麼，你可以預料到會看到以下的輸出：

```
name1 = Makefile
name2 = inc.mk
```

其它的特殊環境變數
GNU make 也支援一個特殊變數，任何設給該變數的值都會被忽略；它總是傳回它的特殊值。

第一個特殊變數是 .VARIABLES 。當展開此變數時，會引入(include)一張在所有到目前爲止的 makefile 中所定義的全域變數名稱的表。這裡頭引入(include)了值爲空白的變數（如內建變數，參考 暗示性規則所使用的變數 ），但不包括只在某個指定標的的內容中定義的變數。

3.5 makefile檔案重新生成的過程

有時makefile檔案可以由其它檔案重新生成，如從RCS或SCCS檔案生成等。如果一個makefile檔案可以從其它檔案重新生成，一定注意讓make更新makefile檔案之後再讀取makefile檔案。

完成讀取所有的makefile檔案之後，make檢查每一個目標，並試圖更新它。如果對於一個makefile檔案有說明它怎樣更新的規則（無論在當前的makefile檔案中或其它makefile檔案中），或者存在一條隱含規則說明它怎樣更新（具體內容參見使用隱含規則），則在必要的時候該makefile檔案將會自動更新。在所有的makefile檔案檢查之後，如果發現任何一個makefile檔案發生變化，make就會清空所有記錄，並重新讀入所有makefile檔案。（然後再次試圖更新這些makefile檔案，正常情況下，因爲這些makefile檔案已被更新，make將不會再更改它們。）

如果您知道您的一個或多個makefile檔案不能重新建立，也許由於執行效率緣故，您不希望make按照隱含規則搜尋或重建它們，您應使用正常的方法阻止按照隱含規則檢查它們。例如，您可以寫一個具體的規則，把這些makefile檔案當作目標，但不提供任何命令（詳細內容參閱使用空命令）。

如果在makefile檔案中指定依據**雙冒號規則(::)**使用命令重建一個檔案，但沒有提供先決條件，則一旦make執行就會重建該檔案（詳細內容參見**雙冒號規則(::)**）。同樣，如果在makefile檔案中指定依據**雙冒號規則(::)**使用命令重建的一個makefile檔案，並且不提供先決條件，則一旦make執行就會重建該makefile檔案，然後重新讀入所有makefile檔案，然後再重建該makefile檔案，再重新讀入所有makefile檔案，如此往複陷入無限循環之中，致使make不能再完成別的任務。如果要避免上述情況的發生，一定注意不要依據**雙冒號規則(::)**使用命令並且不提供先決條件重建任何makefile檔案。

如果您沒有使用‘-f’或‘--file’指定makefile檔案，make將會使用預設的makefile檔案名（詳細內容參見3.2節內容）。不像使用‘-f’或‘--file’選項指定具體的makefile檔案，這時make不能確定makefile檔案是否存在。如果預設的makefile檔案不存在，但可以由執行的make依據規則建立，您需要執行這些規則，建立要使用的makefile檔案。

如果預設的makefile檔案不存在，make將會按照搜尋的次序將它們試著建立，一直到將makefile檔案成功建立或make將所有的檔案名都試過來。注意make不能找到或建立makefile檔案不是錯誤，makefile檔案並不是執行make必須的。

因爲即使您使用‘-t’特別指定，‘-t’或‘--touch’選項對更新makefile檔案不產生任何影響， makefile檔案仍然會更新，所以當您使用‘-t’或‘--touch’選項時，您不要使用舊式的makefile檔案來決定‘touch’哪個目標（具體含義參閱代替執行命令）。同樣，因爲‘-q' (或 ‘--question') 和 ‘-n' (或 ‘--just-print')也能不阻止更新makefile檔案，所以舊式的makefile檔案對其它的目標將產生錯誤的輸出結果。如，‘make -f mfile -n foo’命令將這樣執行：更新‘mfile’，然後讀入，再輸出更新‘foo’的命令和先決條件，但並不執行更新‘foo’，注意，所有回顯的更新‘foo’的命令是在更新後的‘mfile’中指定的。

在實際使用過程中，您一定會遇見確實希望阻止更新makefile檔案的情況。如果這樣，您可以在makefile檔案命令行中將需要更新的makefile檔案指定爲目標，如此則可阻止更新makefile檔案。一旦makefile檔案名被明確指定爲一個目標，選項‘-t’等將會對它發生作用。如這樣設定，‘make -f mfile -n foo’命令將這樣執行：讀入‘mfile’，輸出更新‘foo’的命令和先決條件，但並不執行更新‘foo’。回顯的更新‘foo’的命令引入(include)在現存的‘mfile’中。

3.6 重載其它makefile檔案

有時一個makefile檔案和另一個makefile檔案相近也是很有用的。您可以使用‘include’指令把更多的makefile檔案引入(include)進來，如此可加入更多的目標和定義的變數。然而如果兩個makefile檔案對相同的目標給出了不同的命令，make就會產生錯誤。
在主makefile檔案（要引入(include)其它makefile檔案的那個）中，您可以使用萬用字元樣式規則說明只有在依靠當前makefile檔案中的訊息不能重新建立目標時，make才搜尋其它的makefile檔案，詳細內容參見定義與重新定義樣式規則。例如：如果您有一個說明怎樣建立目標‘foo’（和其它目標）的makefile檔案稱爲‘Makefile’，您可以編寫另外一個稱爲‘GNUmakefile’的makefile檔案引入(include)以下語句：
foo:

```
	frobnicate > foo
%: force
	@$(MAKE) -f Makefile $@
force: ;
```

如果鍵入‘make foo’，make就會找到‘GNUmakefile’，讀入，然後執行‘frobnicate > foo’。如果鍵入‘make bar’，make發現無法根據‘GNUmakefile’建立‘bar’，它將使用樣式規則提供的命令：‘make f Makefile bar’。如果在‘Makefile’中提供了‘bar’更新的規則，make就會使用該規則。對其它‘GNUmakefile’不提供怎樣更新的目標make也會同樣處理。這種工作的模式是使用了樣式規則中的樣式匹配符‘%’，它可以和任何目標匹配。該規則指定了一個先決條件‘force’，用來保證命令一定要執行，無論目標檔案是否存在。我們給出的目標‘force’時使用了空命令，這樣可防止make按照隱含規則搜尋和建立它，否則，make將把同樣的匹配規則應用到目標‘force’本身，從而陷入建立先決條件的循環中。

3.7 make讀取makefile檔案的過程

GNU make把它的工作明顯的分爲兩個階段。在第一階段，make讀取makefile檔案，包括makefile檔案本身、內置變數及其值、隱含規則和具體規則、構造所有目標的依靠圖表和它們的先決條件等。在第二階段，make使用這些內置的組織決定需要重新構造的目標以及使用必要的規則進行工作。

了解make兩階段的工作模式十分重要，因爲它直接影響變數、函數展開模式；而這也是編寫makefile檔案時導致一些錯誤的主要來源之一。下面我們將對makefile檔案中不同架構的展開模式進行總結。我們稱在make工作第一階段發生的展開是立即展開：在這種情況下，make對makefile檔案進行語法分析時把變數和函數直接展開爲架構單元的一部分。我們把不能立即執行的展開稱爲延時(deferred)展開。延時(deferred)展開架構直到它已出現下上下文架構中或make已進入到了第二工作階段時才執行展開。

您可能對這一部分內容不熟悉。您可以先看完後面幾章對這些知識熟悉後再參考本節內容。

變數賦値

變數的定義語法形式如下：

immediate = deferred
immediate ?= deferred
immediate := immediate
immediate += deferred or immediate

define immediate
  deferred
endef

對於附加操作符‘+=’，右邊變數如果在前面使用（:=）定義爲簡單展開變數則是立即變數，其它均爲延時(deferred)變數。

條件語句

整體上講，條件語句都按語法立即分析，常用的有：ifdef、ifeq、ifndef和inneq。

定義規則

規則不論其形式如何，都按相同的模式展開。

immediate : immediate ; deferred
	deferred

目標和先決條件部分都立即展開，用於構造目標的命令通常都是延時(deferred)展開。這個通用的規律對具體規則、樣式規則、後置規則(suffix rule)、靜態樣式規則和簡單先決條件定義都適用。

4編寫規則

makefile檔案中的規則是用來說明何時以及怎樣重建特定檔案的，這些特定的檔案稱為該規則的目標（通常情況下，每個規則只有一個目標）。在規則中列舉的其它檔案稱為目標的先決條件，同時規則還給出了目標建立、更新的命令。一般情況下規則的次序無關緊要，但決定預設最終目標時卻是例外。預設最終目標是您沒有另外指定最終目標時，make認定的最終目標。預設最終目標是makefile檔案中的第一條規則的目標。如果第一條規則有多個目標，只有第一個目標被認為是預設最終目標。有兩種例外的情況：以句點（'.'）開始的目標不是預設最終目標（如果該目標引入(include)一個或多個斜線(/)'/'，則該目標也可能是預設最終目標）；另一種情況是樣式規則定義的目標不是預設最終目標（參閱定義與重新定義樣式規則）。

所以，我們編寫makefile檔案時，通常將第一個規則的目標定為編譯全部程式或是由makefile檔案表述的所有程式（經常設定一個稱為'all'的目標）。參閱指定最終目標的參數。

4.1規則的語法

通常一條規則形式如下：

```
targets : prerequisites
	command
	...
```

或：

```
targets : prerequisites ; command
	command
	...
```

目標（target）是檔案的名稱，中間由空格隔開。萬用字元可以在檔案名中使用（參閱在檔案名中使用萬用字元），'a（m）' 形式的檔案名表示成員m在檔案a中（參閱資料庫成員目標）。一般情況下，一條規則只有一個目標，但偶爾由於其它原因一條規則有多個目標（參閱具有多個目標的規則）。

命令行以Tab字符開始，第一個命令可以和先決條件在一行，命令和先決條件之間用分號隔開，也可以在先決條件下一行，以Tab字符為行的開始。這兩種方法的效果一樣，參閱在規則中使用命令。
因為美元符號已經用為變數引用的開始符，如果您真希望在規則中**使用美元符號**，您必須連寫兩次，**'$$'**（參閱使用變數）。您可以把一長行在中間插入'\'使其分為兩行，也就是說，一行的尾部是'\'的話，表示下一行是本行的繼續行。但這並不是必須的，make沒有對makefile檔案中行的長度進行限制。一條規則可以告訴make兩件事情：何時目標已經過時，以及怎樣在必要時更新它們。

判斷目標舊式的準則和先決條件關係密切，先決條件也由檔案名構成，檔案名之間由空格隔開，萬用字元和資料庫成員也允許在先決條件中出現。一個目標如果不存在或它比其中一個先決條件的修改時間早，則該目標已經過時。該思想來源於目標是根據先決條件的訊息計算得來的，因此一旦任何一個先決條件發生變化，目標檔案也就不再有效。目標的更新模式由命令決定。命令由shell解釋執行，但也有一些另外的特點。參閱在規則中使用命令。

4.2 在檔案名中使用萬用字元

一個簡單的檔案名可以透過使用萬用字元代表許多檔案。**Make中的萬用字元和Bourne shell中的萬用字元一樣是'*'、'?'和'[…]'**。例如：'*.C'指在當前目錄中所有以'.C'結尾的檔案。

字符'~'在檔案名的前面也有特殊的含義。**如果字符'~'單獨或後面跟一個斜線(/)'/'，則代表您的home目錄。如'~/bin'展開為'/home/bin'。如果字符'~'後面跟一個字，它展開為home目錄下以該字為名字的目錄，如'~John/bin'表示'home/John/bin'**。在一些作業系統（如ms-dos，ms-windows）中不存在home目錄，可以透過設定環境變數home來類比。

在目標、先決條件和命令中的萬用字元自動展開。在其它上下文中，萬用字元只有在您明確表明呼叫萬用字元函數時才展開。

萬用字元另一個特點是如果萬用字元前面是反斜線(\)‘\’，則該萬用字元失去通配能力。如‘foo*bar’表示一個特定的檔案其名字由‘foo’、‘*’和‘bar’構成。

4.2.1萬用字元例子

萬用字元可以用在規則的命令中，此時萬用字元由shell展開。例如，下面的規則刪除所有OBJ檔案：

```
clean：
        rm  -f  *.o
```

萬用字元在規則的先決條件中也很有用。在下面的makefile規則中，‘make print’將列印所有從上次您列印以後又有改動的‘.c’檔案：

```
print: *.c
    lpr -p $?
    touch print
```

本規則使用‘ptint’作爲一個空目標檔案（參看使用空目標檔案記錄事件）；自動變數‘$?’用來列印那些已經修改的檔案，參看自動變數。

當您定義一個變數時萬用字元不會展開，如果您這樣寫：

```
objects = *.o
```

變數objects的值實際就是字元串‘*.o’。然而，如果您在一個目標、先決條件和命令中使用變數objects的值，萬用字元將在那時展開。使用下面的語句可使萬用字元展開：

```
objects=$(wildcard *.o)
```

詳細內容參閱函數wildcard。

4.2.2使用萬用字元的常見錯誤

下面有一個幼稚使用萬用字元展開的例子，但實際上該例子不能完成您所希望完成的任務。假設可執行檔案‘foo’由在當前目錄的所有OBJ檔案建立，其規則如下：

```
objects = *.o

foo : $(objects)
    cc -o foo $(CFLAGS) $(objects)
```

由於變數objects的值爲字元串‘*.o’，萬用字元在目標‘foo’的規則下展開，所以每一個OBJ檔案都會變爲目標‘foo’的先決條件，並在必要時重新編譯自己。

但如果您已刪除了所有的OBJ檔案，情況又會怎樣呢？因沒有和萬用字元匹配的檔案，所以目標‘foo’就依靠了一個有著奇怪名字的檔案‘*.o’。因爲目錄中不存在該檔案，make將發出不能建立‘*.o’的錯誤訊息。這可不是所要執行的任務。

實際上，使用萬用字元獲得正確的結果是可能的，但您必須使用稍微複雜一點的技術，該技術包括使用函數wildcard和替代字元串等。詳細內容將在下一節論述。

微軟的作業系統（MS-DOS、MS-WINDOWS）使用反斜線(\)分離目錄路徑(stem)，如：

C:\foo\bar\bar.c

這和Unix風格‘c:/foo/bar/bar.c’相同（‘c:’是驅動器字母）。當make在這些系統上執行時，不但支援在路徑(stem)中存在反斜線(\)也支援Unix風格的前斜線(/)。但是這種對反斜線(\)的支援不包括萬用字元展開，因爲萬用字元展開時，反斜線(\)用作引用字符。所以，在這些場合您必須使用Unix風格的前斜線(/)。

4.2.3函數wildcard(萬用字元)

萬用字元在規則中可以自動展開，但設定在變數中或在函數的參數中萬用字元一般不能正常展開。如果您需要在這些場合展開萬用字元，您應該使用函數wildcard，格式如下：

```
$(wildcard pattern...)
```

可以在makefile檔案的任何地方使用該字元串，應用時該字元串被一列在指定目錄下存在的並且檔案名和給出的檔案名

的格式相符合的檔案所代替，檔案名中間由空格隔開。如果沒有和指定格式一致的檔案，則函數wildcard的輸出將會省略。注意這和在規則中萬用字元展開的模式不同，在規則中使用逐字展開模式，而不是省略模式（參閱上節）。
使用函數wildcard得到指定目錄下所有的C語言源程式檔案名的命令格式爲：
$(wildcard *.c)

我們可以把所獲得的C語言源程式檔案名的字元串透過將‘.c’後置變爲‘.o’轉換爲OBJ檔案名的字元串，其格式爲

$(patsubst %.c,%.o,$(wildcard *.c))
這裡我們使用了另外一個函數：patsubst，詳細內容參閱字元串替換和分析函數。
這樣，一個編譯特定目錄下所有C語言源程式並把它們連接在一起的makefile檔案可以寫成如下格式：

```
objects := $(patsubst %.c,%.o,$(wildcard *.c))

foo : $(objects)
      cc -o foo $(objects)
```

這裡使用了編譯C語言源程式的隱含規則，因此沒有必要爲每個檔案寫具體編譯規則。‘:=’是‘=’的變異，對‘:=’的解釋，參閱兩種風格的變數。

4.3在目錄中搜尋先決條件

對於大型系統，把源檔案安放在一個單獨的目錄中，而把二進制檔案放在另一個目錄中是十分常見的。Make的目錄搜尋特性使自動在幾個目錄搜尋先決條件十分容易。當您在幾個目錄中重新安排您的檔案，您不必改動單獨的規則，僅僅改動一下搜尋路徑(stem)即可。

4.3.1 VPATH：所有先決條件的搜尋路徑(stem)

make變數VPATH的值指定了make搜尋的目錄。經常用到的是那些引入(include)先決條件的目錄，並不是當前的目錄；但VPATH指定了make對所有檔案都適用的目錄搜尋序列，包括了規則的目標所需要的檔案。
如果一個作爲目標或先決條件的檔案在當前目錄中不存在，make就會在VPATH指定的目錄中搜尋該檔案。如果在這些目錄中找到要尋找的檔案，則就像這些檔案在當前目錄下存在一樣，規則把這些檔案指定爲先決條件。參閱編寫搜尋目錄的shell命令。

在VPATH變數定義中，目錄的名字由冒號或空格分開。目錄列舉的次序也是make 搜尋的次序。在MS-DOS、MS-WINDOWS系統中，VPATH變數定義中的目錄的名字由分號分開，因爲在這些系統中，冒號用爲路徑(stem)名的一部分（通常在驅動器字母後面）。例如：

```
VPATH = src:../headers
```

指定了兩個目錄，‘src’和‘…/headers’，make也按照這個次序進行搜尋。使用該VPATH的值，下面的規則，

```
foo.o : foo.c
```

在執行時就像如下寫法一樣會被中斷：

```
foo.o : src/foo.c
```

然後在src目錄下搜尋foo.c。

4.3.2 vpath指令

vpath指令（注意字母是小寫）和VPATH變數類似，但卻更具靈活性。vpath指令允許對符合一定格式類型的檔案名指定一個搜尋路徑(stem)。這樣您就可以對一種格式類型的檔案名指定一個搜尋路徑(stem)，對另外格式類型的檔案名指定另外一個搜尋路徑(stem)。總共由三種形式的vpath指令：

vpath pattern directories

對一定格式類型的檔案名指定一個搜尋路徑(stem)。搜尋的路徑(stem)由一列要搜尋的目錄構成，目錄由冒號（在MS-DOS、MS-WINDOWS系統中用分號）或空格隔開，和VPATH變數定義要搜尋的路徑(stem)格式一樣。

vpath pattern

清除和一定類型格式相聯繫的搜尋路徑(stem)。

vpath

清除所有前面由vapth指令指定的搜尋路徑(stem)。

一個vpath的格式pattern是一個包含一個’%’的字元串。該字元串必須和正搜尋的一個先決條件的檔案名匹配，字符%可和任何字元串匹配（關於樣式規則，參閱定義與重新定義樣式規則）。例如，%.h和任何檔案名以.h結尾的檔案匹配。如果不使用‘%’，格式必須與先決條件精確匹配，這種情況很少使用。
在vpath指令格式中的字符‘%’可以透過前面的反斜線(\)被引用。引用其它字符‘%’的反斜線(\)也可以被更多的反斜線(\)引用。引用字符‘%’和其它反斜線(\)的反斜線(\)在和檔案名比較之前和格式是分開的。如果反斜線(\)所引用的字符‘%’沒有錯誤，則該反斜線(\)不會執行帶來任何危害。
如果vpath指令格式和一個先決條件的檔案名匹配，並且在當前目錄中該先決條件不存在，則vpath指令中指定的目錄和VPATH變數中的目錄一樣可以被搜尋。例如：
vpath %.h ../headers
將告訴make如果在當前目錄中以‘.h’結尾檔案不存在，則在‘../headers’目錄下搜尋任何以‘.h’結尾先決條件。

如果有幾個vpath指令格式和一個先決條件的檔案名匹配，則make一個接一個的處理它們，搜尋所有在指令中指定的目錄。Make按它們在makefile檔案中出現的次序控制多個vpath指令，多個指令雖然有相同的格式，但它們是相互獨立的。以下代碼：

```
vpath %.c foo
vpath %   blish
vpath %.c bar
```

表示搜尋`.c'檔案先搜尋目錄`foo'、然後`blish'，最後`bar'；如果是如下代碼：

```
vpath %.c foo:bar
vpath %   blish
```

表示搜尋`.c'檔案先搜尋目錄‘foo'、然後‘bar'，最後‘blish'。

4.3.3目錄搜尋過程

當透過目錄搜尋找到一個檔案，該檔案有可能不是您在先決條件清單中所列出的先決條件；有時透過目錄搜尋找到的路徑(stem)也可能被廢棄。Make決定對透過目錄搜尋找到的路徑(stem)儲存或廢棄所依據的算法如下：
1、如果一個目標檔案在makefile檔案所在的目錄下不存在，則將會執行目錄搜尋。
2、如果目錄搜尋成功，則路徑(stem)和所得到的檔案暫時作為目標檔案儲存。
3、所有該目標的先決條件用相同的方法考察。
4、把先決條件處理完成後，該目標可能需要或不需要重新建立：
1、如果該目標不需要重建，目錄搜尋時所得到的檔案的路徑(stem)用作該目標所有先決條件的路徑(stem)，同時包含該目標檔案。簡而言之，如果make不必重建目標，則您使用透過目錄搜尋得到的路徑(stem)。
2、如果該目標需要重建，目錄搜尋時所得到的檔案的路徑(stem)將廢棄，目標檔案在makefile檔案所在的目錄下重建。
簡而言之，如果make要重建目標，是在makefile檔案所在的目錄下重建目標，而不是在目錄搜尋時所得到的檔案的路徑(stem)下 。

該算法似乎比較複雜，但它卻可十分精確的解釋實際您所要的東西。
其它版本的make使用一種比較簡單的算法：如果目標檔案在當前目錄下不存在，而它透過目錄搜尋得到，不論該目標是否需要重建，始終使用透過目錄搜尋得到的路徑(stem)。
實際上，如果在GNU make中使您的一些或全部目錄具備這種行為，您可以使用GPATH變數來指定這些目錄。
GPATH變數和VPATH變數具有相同的語法和格式。如果透過目錄搜尋得到一個舊式的目標，而目標存在的目錄又出現下GPATH變數，則該路徑(stem)將不廢棄，目標將在該路徑(stem)下重建。

4.3.4編寫目錄搜尋的shell命令

即使透過目錄搜尋在其它目錄下找到一個先決條件，不能改變規則的命令，這些命令同樣按照原來編寫的模式執行。因此，您應該小心的編寫這些命令，以便它們可以在make能夠在發現先決條件的目錄中處理先決條件。

借助諸如‘$^’的自動變數可更好的使用shell命令（參閱自動變數）。例如，**‘$^’的值代表所有的先決條件清單，並引入(include)尋找先決條件的目錄；‘$@’的值是目標**。

```
foo.o : foo.c
```

```
    cc -c $(CFLAGS) $^ -o $@
```

變數CFLAGS存在可以方便您利用隱含規則指定編譯C語言源程式的旗標。我們這裡使用它是爲了保持編譯C語言源程式一致性。參閱隱含規則使用的變數。
先決條件通常情況下也引入(include)頭檔案，因自動變數‘**$<’的值是第一個先決條件**，因此這些頭檔案您可以不必在命令中提及，

例如：

```
VPATH = src:../headers
foo.o : foo.c defs.h hack.h
    cc -c $(CFLAGS) $< -o $@
```

4.3.5 目錄搜尋和隱含規則

搜尋的目錄是由變數VPATH或隱含規則引入的vpath指令指定的（詳細參閱使用隱含規則）。例如，如果檔案‘foo.o’沒有具體的規則，make則使用隱含規則：如檔案foo.c存在，make使用內置的規則編譯它；如果檔案foo.c不在當前目錄下，就搜尋適當的目錄，如在別的目錄下找到foo.c，make同樣使用內置的規則編譯它。
隱含規則的命令使用自動變數是必需的，所以隱含規則可以自然地使用目錄搜尋得到的檔案。

4.3.6 連接庫(Link Libraries)的搜尋目錄

對於連接庫(Link Libraries)檔案，目錄搜尋採用一種特別的模式。這種特別的模式開始於：您寫一個先決條件，它的名字是‘-lname’的形式。（您可以在這裡寫一些奇特的字符，因爲先決條件正常是一些檔案名，庫檔案名通常是‘libname.a’ 的形式，而不是‘-lname’ 的形式。）
當一個先決條件的名字是‘-lname’的形式時，make特別地在當前目錄下、與vpath匹配的目錄下、VPATH指定的目錄下以及‘/lib’, ‘/usr/lib', 和 ‘prefix/lib'（正常情況爲`/usr/local/lib'，但是MS-DOS、MS-Windows版本的make的行爲好像是prefix定義爲DJGPP安裝樹的根目錄的情況）目錄下搜尋名字爲‘libname.so'的檔案然後再處理它。如果沒有搜尋到‘libname.so'檔案，然後在前述的目錄下搜尋‘libname.a'檔案。

例如，如果在您的系統中有‘/usr/lib/libcurses.a'的庫檔案，則：

```
foo : foo.c -lcurses
    cc $^ -o $@
```

如果‘foo’比‘foo.c’更舊，將導致命令‘cc foo.c /usr/lib/libcurses.a -o foo'執行。

預設情況下是搜尋‘libname.so' 和‘libname.a'檔案，具體搜尋的檔案及其類型可使用.LIBPATTERNS變數指定，這個變數值中的每一個字都是一個字元串格式。當尋找名爲‘-lname’的先決條件時，make首先用name替代清單中第一個字中的格式部分形成要搜尋的庫檔案名，然後使用該庫檔案名在上述的目錄中搜尋。 如果沒有發現庫檔案，則使用清單中的下一個字，其餘以此類推。
.LIBPATTERNS變數預設的值是"‘lib%.so lib%.a'"，該值對前面描述的預設行爲提供支援。您可以透過將該值設爲空值從而徹底關閉對連接庫(Link Libraries)的展開。

4.4假想(phony)目標

假想(phony)目標並不是一個真正的檔案名，它僅僅是您製定的一個具體規則所執行的一些命令的名稱。使用假想(phony)目標有兩個原因：避免和具有相同名稱的檔案衝突和改善性能。
如果您寫一個其命令不建立目標檔案的規則，一旦由於重建而提及該目標，則該規則的命令就會執行。

這裡有一個例子：

```
clean:
    rm *.o temp
```

因爲rm命令不建立名爲‘clean’的檔案，所以不應有名爲‘clean’的檔案存在。因此不論何時您發布`make clean'指令，rm命令就會執行。

假想(phony)目標能夠終止任何在目錄下建立名爲‘clean’的檔案工作。但如在目錄下存在檔案clean，因爲該目標clean沒有先決條件，所以檔案clean始終會認爲已經該更新，因此它的命令將永不會執行。爲了避免這種情況，您應該使用像如下特別的**.PHONY目標樣式將該目標具體的聲明爲一個假想(phony)目標**：

.PHONY : clean
一旦這樣聲明，‘make clean’命令無論目錄下是否存在名爲‘clean’的檔案，該目標的命令都會執行。
因爲make知道假想(phony)目標不是一個需要根據別的檔案重新建立的實際檔案，所以它將跳過隱含規則搜尋假想(phony)目標的步驟（詳細內容參閱使用隱含規則）。這是把一個目標聲明爲假想(phony)目標可以提升執行效率的原因，因此使用假想(phony)目標您不用擔心在目錄下是否有實際檔案存在。這樣，對前面的例子可以用假想(phony)目標的寫出，其格式如下：

```
.PHONY: clean
clean:
        rm *.o temp
```

另外一個使用假想(phony)目標的例子是使用make的遞迴進行連接的情況：此時，makefile檔案常常引入(include)列舉一系列需要建立的子目錄的變數。不用假想(phony)目標完成這種任務的方法是使用一條規則，其命令是一個在各個子目錄下循環的shell命令，

如下面的例子：

```
subdirs:
        for dir in $(SUBDIRS); do \
          $(MAKE) -C $$dir; \
        done
```

但使用這個方法存在下述問題：首先，這個規則在建立子目錄時產生的任何錯誤都不及時發現，因此，當一個子目錄建立失敗時，該規則仍然會繼續建立剩餘的子目錄。雖然該問題可以添加監視錯誤產生並退出的shell命令來解決，但非常不幸的是如果make使用了‘-k’選項，這個問題仍然會產生。第二，也許更重要的是您使用了該方法就失去使用make並行處理的特點能力。

使用假想(phony)目標（如果一些子目錄已經存在，您則必須這樣做，否則，不存在的子目錄將不會建立）則可以避免上述問題：

```
SUBDIRS = foo bar baz

.PHONY: subdirs $(SUBDIRS)

subdirs: $(SUBDIRS)

$(SUBDIRS):
        $(MAKE) -C $

foo: baz
```

此時，如果子目錄‘baz’沒有建立完成，子目錄’foo’將不會建立；當試圖使用並行建立時這種關係的聲明尤其重要。

一個假想(phony)目標不應該是一個實際目標檔案的先決條件，如果這樣，make每次執行該規則的命令，目標檔案都要更新。只要假想(phony)目標不是一個真實目標的先決條件，假想(phony)目標的命令只有在假想(phony)目標作爲特別目標時才會執行（參閱指定最終目標的參數）。
假想(phony)目標也可以有先決條件。當一個目錄下引入(include)多個程式時，使用假想(phony)目標可以方便的在一個makefile檔案中描述多個程式的更新。重建的最終目標預設情況下是makefile檔案的第一個規則的目標，但將多個程式作爲假想(phony)目標的先決條件則可以輕鬆的完成在一個makefile檔案中描述多個程式的更新。如下例：

```
all : prog1 prog2 prog3
.PHONY : all

prog1 : prog1.o utils.o
        cc -o prog1 prog1.o utils.o
```

```
prog2 : prog2.o
        cc -o prog2 prog2.o

prog3 : prog3.o sort.o utils.o
        cc -o prog3 prog3.o sort.o utils.o
```

這樣，您可以重建所有程式，也可以參數的形式重建其中的一個或多個（如‘make prog1 prog3'）。
當一個假想(phony)目標是另一個假想(phony)目標的先決條件，則該假想(phony)目標將作爲一個假想(phony)目標的子例程。例如，這裡‘make cleanall'用來刪除OBJ檔案、diff檔案和程式檔案：

```
.PHONY: cleanall cleanobj cleandiff

cleanall : cleanobj cleandiff
        rm program

cleanobj :
        rm *.o

cleandiff :
        rm *.diff
```

4.5 沒有命令或先決條件的規則

如果一個規則沒有先決條件、也沒有命令，而且這個規則的目標也不是一個存在的檔案，則make認爲只要該規則執行，該目標就已被更新。這意味著，所有以這種規則的目標爲先決條件的目標，它們的命令將總被執行。這裡舉一個例子：

```
clean: FORCE
        rm $(objects)
FORCE:
```

這裡的目標‘FORCR’滿足上面的特殊條件，所以以其爲先決條件的目標‘clean’將總強製它的命令執行。關於‘FORCR’的名字沒有特別的要求，但‘FORCR’是習慣使用的名字。

也許您已經明白，使用‘FORCR’的方法和使用假想(phony)目標（.PHONY: clean）的結果一樣，但使用假想(phony)目標更具體更靈活有效，由於一些別的版本的make不支援假想(phony)目標，所以‘FORCR’出現下許多makefile檔案中。參閱假想(phony)目標。

4.6使用空目標檔案記錄事件

空目標是一個假想(phony)目標變數，它用來控制一些命令的執行，這些命令可用來完成一些經常需要的具體任務。但又不像真正的假想(phony)目標，它的目標檔案可以實際存在，但檔案的內容與此無關，通常情況下，這些檔案沒有內容。

空目標檔案的用途是用來記錄規則的命令最後一次執行的時間，也是空目標檔案最後更改的時間。它之所以能夠這樣執行是因爲規則的命令中有一條用於更新目標檔案的‘touch’命令。另外，空目標檔案應有一些先決條件（否則空目標檔案沒有存在的意義）。如果空目標比它的先決條件舊，當您命令重建空目標檔案時，有關的命令才會執行。下面有一個例子：

```
print: foo.c bar.c
        lpr -p $?
        touch print
```

按照這個規則，如果任何一個源檔案從上次執行‘make print'以來發生變化，鍵入‘make print'則執行lpr命令。自動變數‘$?’用來列印那些發生變化的檔案（參閱自動變數）。

4.7 內建的特殊目標名

- 一些名字作爲目標使用則含有特殊的意義：
- .PHONY
- 特殊目標.PHONY的先決條件是假想(phony)目標。假想(phony)目標是這樣一些目標，make無條件的執行它命令，和目錄下是否存在該檔案以及它最後一次更新的時間沒有關係。詳細內容參閱假想(phony)目標。
- .SUFFIXES
- 特殊目標.SUFFIXES的先決條件是一列用於後置規則(suffix rule)檢查的後置。詳細內容參閱舊式的後置規則(suffix rule)。
- .DEFAULT
- .DEFAULT指定一些命令，這些命令用於那些沒有找到規則（具體規則或隱含規則）更新的目標。詳細內容參閱定義最新類型的-預設規則。如果.DEFAULT指定了一些命令，則所有提及到的檔案只能作爲先決條件，而不能作爲任何規則的目標；這些指定的命令也只按照他們自己的模式執行。詳細內容參閱隱含規則搜尋算法。
- .PRECIOUS
- 特殊目標.PRECIOUS的先決條件將按照下面給定的特殊模式進行處理：如果在執行這些目標的命令的過程中，make被關閉或中斷，這些目標不能被刪除，詳細內容參閱關閉和中斷make；如果目標是中間檔案，即使它已經沒有任何用途也不能被刪除，具體情況和該目標正常完成一樣，參閱隱含規則鏈；該目標的其它功能和特殊目標.SECONDARY的功能重疊。如果規則的目標樣式與先決條件的檔案名匹配，您可以使用隱含規則的格式（如‘%.O’）列舉目標作爲特殊目標.PRECIOUS的先決條件檔案來儲存由這些規則建立的中間檔案。
- .INTERMEDIATE
- 特殊目標.INTERMEDIATE的先決條件被處理爲中間檔案。詳細內容參見隱含規則鏈。.INTERMEDIATE如果沒有先決條件檔案，它將不會發生作用。
- .SECONDARY
- 特殊目標.SECONDARY的先決條件被處理爲中間檔案，但它們永遠不能自動刪除。詳細內容參見隱含規則鏈。.SECONDARY如果沒有先決條件檔案，則所有的makefile檔案中的目標都將被處理爲中間檔案。
- .DELETE_ON_ERROR
- 如果在makefile檔案的某處.DELETE_ON_ERROR作爲一個目標被提及，則如果該規則發生變化或它的命令沒有正確完成而退出，make將會刪除該規則的目標，具體行爲和它受到了刪除信號一樣。詳細內容參閱命令錯誤。
- .IGNORE
- 如果您特別爲目標.IGNORE指明先決條件，則MAKE將會忽略處理這些先決條件檔案時執行命令產生的錯誤。如果.IGNORE作爲一個沒有先決條件的目標提出來，MAKE將忽略處理所有檔案時產生的錯誤。.IGNORE命令並沒有特別的含義，.IGNORE的用途僅是爲了和早期版本的兼容。因爲.IGNORE影響所有的命令，所以它的用途不大；我們推薦您使用其它方法來忽略特定命令產生的錯誤。詳細內容參閱命令錯誤。
- .SILENT
- 如果您特別爲.SILENT指明先決條件，則在執行之前MAKE將不會回顯重新構造檔案的命令。如果.SILENT作爲一個沒有先決條件的目標提出來，任何命令在執行之前都不會列印。.SILENT並沒有特別的含義，其用途僅是爲了和早期版本的兼容。我們推薦您使用其它方法來處理那些不列印的命令。詳細內容參閱命令回顯。如果您希望所有的命令都不列印，請使用‘-s’或‘-silent’選項(詳細參閱選項概要)。
- .EXPORT_ALL_VARIABLES
- 如該特殊目標簡單的作爲一個目標被提及，MAKE將預設地把所有變數都傳遞到子進程中。參閱使與子MAKE通信的變數。
- .NOTPARALLEL

如果.NOTPARALLEL作爲一個目標提及，即使給出‘-j’選項，make也不使用並行執行。但遞迴的make命令仍可並行執行（在呼叫的makefile檔案中引入(include).NOTPARALLEL的目標的例外）。.NOTPARALLEL的任何先決條件都將忽略。

任何定義的隱含規則後置如果作爲目標出現都會視爲一個特殊規則，即使兩個後置串聯起來也是如此，例如‘.c.o’。這些目標稱爲後置規則(suffix rule)，這種定義方法是舊式的定義隱含規則的方法（目前仍然廣泛使用的方法）。原則上，如果您要把它分爲兩個並把它們加到後置清單中，任何目標名都可採用這種方法指定。實際上，後置一般以‘.’開始，因此，這些特別的目標同樣以‘.’開始。具體參閱舊式的後置規則(suffix rule)。

4.8 具有多個目標的規則

具有多個目標的規則等同於寫多條規則，這些規則除了目標不同之外，其餘部分完全相同。相同的命令應用於所有目標，但命令執行的結果可能有所差異，因此您可以在命令中使用‘$@’分發不同的實際目標名稱。這條規則同樣意味

著所有的目標有相同的先決條件。

- 在以下兩種情況下具有多個目標的規則相當有用：
  您僅僅需要依賴，但不需要任何命令。例如：

kbd.o command.o files.o: command.h

- 爲三個提及的目標檔案給出附加的共同先決條件。
  所有的目標使用相同的命令。但命令的執行結果未必完全相同，因爲自動變數‘$@’可以在重建時指定目標（參閱自動變數）。例如：

```
bigoutput littleoutput : text.g
        generate text.g -$(subst output,,$@) > $@
```

等同於：

```
bigoutput : text.g
        generate text.g -big > bigoutput
littleoutput : text.g
        generate text.g -little > littleoutput
```

這裡我們假設程式可以產生兩種輸出檔案類型：一種給出‘-big’，另一種給出‘-little’。參閱字元串代替和分析函數，對函數subst的解釋。
如果您喜歡根據目標變換先決條件，像使用變數**‘$@’變換命令**一樣。您不必使用具有多個目標的規則，您可以使用靜態樣式規則。詳細內容見下文。

4.9 具有多條規則的目標

一個目標檔案可以有多個規則。在所有規則中提及的先決條件都將融合在一個該目標的先決條件清單中。如果該目標比任何一個先決條件‘舊’，所有的命令將執行重建該目標。
但如果一條以上的規則對同一檔案給出多條命令，make將使用最後給出的規則，同時列印錯誤訊息。（作爲特例，如果檔案名以點‘.’開始，不列印出錯誤訊息。這種古怪的行爲僅僅是爲了和其它版本的make兼容）。您沒有必要這樣編寫您的makefile檔案，這正是make給您發出錯誤訊息的原因。

一條特別的先決條件規則可以用來立即給多條目標檔案提供一些額外的先決條件。例如，使用名爲‘objects’的變數，該變數包含系統產生的所有輸出檔案清單。如果‘congfig.h’發生變化所有的輸出檔案必須重新編譯，可以採用下列簡單的方法編寫：

```
objects = foo.o bar.o
foo.o : defs.h
bar.o : defs.h test.h
$(objects) : config.h
```

這些可以自由插入或取出而不影響實際指定的目標檔案生成規則，如果您希望斷斷續續的爲目標添加先決條件，這是非常方便的方法。
另外一個添加先決條件的方法是定義一個變數，並將該變數作爲make命令的參數使用。詳細內容參閱變數重載。

例如：

```
extradeps=
$(objects) : $(extradeps)
```

命令`make extradeps=foo.h'含義是將‘foo.h’作爲所有OBJ檔案的先決條件，如果僅僅輸入‘make’命令則不是這樣。
如果沒有具體的規則爲目標的生成指定命令，那麼make將搜尋合適的隱含規則進而確定一些命令來完成生成或重建目標。詳細內容參閱使用隱含規則。

4.10 靜態樣式規則Static Pattern Rules

靜態樣式規則是指定多個目標並能夠根據每個目標名構造對應的先決條件名的規則。靜態樣式規則在用於多個目標時比平常的規則更常用，因爲目標可以不必有完全相同的先決條件；也就是說，這些目標的先決條件必須類似，但不必完全相同。

4.10.1 靜態樣式規則的語法

這裡是靜態樣式規則的語法格式：

```
targets ...: target-pattern: dep-patterns ...
        commands
        ...
```

目標清單指明該規則應用的目標。目標可以含有萬用字元，具體使用和平常的目標規則基本一樣（參閱在檔案名中使用萬用字元）。

目標的格式和先決條件的格式是說明如何計算每個目標先決條件的方法。從匹配目標樣式的目標名中依據格式抽取部分字元串，這部分字元串稱爲徑(stem)。將徑(stem)分發到每一個先決條件格式中產生先決條件名。

每一個格式通常包含字符‘%’。目標樣式匹配目標時，‘%’可以匹配目標名中的任何字元串；這部分匹配的字元串稱爲徑(stem)；剩下的部分必須完全相同。如目標‘foo.o’匹配樣式‘%.o’，字元串‘foo’稱爲徑(stem)。而目標‘foo.c’和‘foo.out’不匹配樣式。

每個目標的先決條件名是使用徑(stem)代替各個先決條件中的‘%’產生。如，如果一個先決條件格式爲‘%.c’，把徑(stem)‘foo’代替先決條件格式中的‘%’生成先決條件的檔案名‘foo.c’。在先決條件格式中不包含‘%’也是合法的，此時對所有目標來說，先決條件是相同的。

在樣式規則中字符‘%’可以用前面加反斜線(\)‘\’方法引用。引用‘%’的反斜線(\)也可以由更多的反斜線(\)引用。引用‘%’、‘\’的反斜線(\)在和檔案名比較或由徑(stem)代替它之前從格式中移走。反斜線(\)不會因爲引用‘%’而混亂。如，格式`the\%weird\\%pattern\\'是`the%weird\' 加上字符‘%'，後面再和字元串 ‘pattern\\'連接。最後的兩個反斜線(\)由於不能影響任何統配符‘%’所以保持不變。

這裡有一個例子，它將對應的‘.c’檔案編譯成‘foo.o’和‘bar.o’。

```
objects = foo.o bar.o
all: $(objects)
$(objects): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
```

這裡‘*$<*’*是自動變數，控制先決條件的名稱，*‘*$@*’*也是自動變數，掌握目標的名稱*。詳細內容參閱自動變數。

每一個指定目標必須和目標樣式匹配，如果不符則產生警告。如果您有一列檔案，僅有其中的一部分和樣式匹配，您可以使用filter函數把不符合的檔案移走（參閱字元串替代和分析函數）：

```
files = foo.elc bar.o lose.o

$(filter %.o,$(files)): %.o: %.c
        $(CC) -c $(CFLAGS) $< -o $@
$(filter %.elc,$(files)): %.elc: %.el
        emacs -f batch-byte-compile $<
```

在這個例子中，‘$(filter %.o,$(files))'的結果是‘bar.o lose.o'，第一個靜態樣式規則是將相應的C語言源檔案編譯更新爲OBJ檔案，‘$(filter %.elc,$(files))' 的結果是‘foo.elc'，它由‘foo.el’構造。

另一個例子是闡明怎樣在靜態樣式規則中使用‘$*’：

```
bigoutput littleoutput : %output : text.g
        generate text.g -$* > $@
```

當命令generate執行時，$*展開爲徑(stem)，即‘big’或‘little’二者之一。

4.10.2靜態樣式規則和隱含規則

靜態樣式規則和定義爲樣式規則的隱含規則有很多相同的地方（詳細參閱定義與重新定義樣式規則）。雙方都有對目標的格式和構造先決條件名稱的格式，差異是make使用它們的時機不同。

隱含規則可以應用於任何於它匹配的目標，但它僅僅是在目標沒有具體規則指定命令以及先決條件可以被搜尋到的情況下應用。如果有多條隱含規則適合，僅有執行其中一條規則，選擇依據隱含規則的定義次序。

相反，靜態樣式規則用於在規則中指明的目標。它不能應用於其它任何目標，並且它的使用模式對於各個目標是固定不變的。如果使用兩個帶有命令的規則發生衝突，則是錯誤。

- 靜態樣式規則因爲如下原因可能比隱含規則更好：
- 對一些文件名不能按句法分類的但可以給出列表的文件，使用靜態樣式規則可以重載隱含規則鏈。
  如果不能精確確定使用的路徑(stem)，您不能確定一些無關緊要的文件是否導致make使用錯誤的隱含規則（因爲隱含規則的選擇根據其定義次序）。使用靜態樣式規則則沒有這些不確定原素：每一條規則都精確的用於指定的目標上。

4.11雙冒號規則(::)

雙冒號規則(::)是在目標名後使用‘：：’代替‘：’的規則。當同一個目標在一條以上的規則中出現時，**雙冒號規則(::)**和平常的規則處理有所差異。

當一目標在多條規則中出現時，所有的規則必須是同一類型：要么都是**雙冒號規則(::)**，要么都是普通規則。如果他們都是**雙冒號規則(::)**，則它們之間都是相互獨立的。如果目標比一個**雙冒號規則(::)**的先決條件‘舊’，則該**雙冒號規則(::)**的命令將執行。這可導致具有同一目標**雙冒號規則(::)**全部或部分執行。

雙冒號規則(::)實際就是將具有相同目標的多條規則相互分離，每一條**雙冒號規則(::)**都獨立的執行，就像這些規則的目標不同一樣。

對於一個目標的**雙冒號規則(::)**按照它們在makefile檔案中出現的順序執行。然而**雙冒號規則(::)**真正有意義的場合是**雙冒號規則(::)**和執行順序無關的場合。

雙冒號規則(::)有點模糊難以理解，它僅僅提供了一種在特定情況下根據引起更新的先決條件檔案不同，而採用不同模式更新目標的機製。實際應用**雙冒號規則(::)**的情況非常罕見。
每一個**雙冒號規則(::)**都應該指定命令，如果沒有指定命令，則會使用隱含規則。詳細內容參閱使用隱含規則。

4.12 自動生成先決條件

在爲一個程式編寫的makefile檔案中，常常需要寫許多僅僅是說明一些OBJ檔案依靠頭檔案的規則。例如，如果‘main.c’透過一條#include語句使用‘defs.h’，您需要寫入下的規則：
main.o: defs.h

您需要這條規則讓make知道如果‘defs.h’一旦改變必須重新構造‘main.o’。由此您可以明白對於一個較大的程式您需要在makefile檔案中寫很多這樣的規則。而且一旦添加或去掉一條#include語句您必須十分小心地更改makefile檔案。
爲避免這種煩惱，現代C編譯器根據原程式中的#include語句可以爲您編寫這些規則。如果需要使用這種功能，通常可在編譯源程式時加入‘-M’開關，

例如，下面的命令：
cc -M main.c
產生如下輸出：
main.o : main.c defs.h

這樣您就不必再親自寫這些規則，編譯器可以爲您完成這些工作。
注意，由於在makefile檔案中提及構造‘main.o’，因此‘main.o’將永遠不會被隱含規則認爲是中間檔案而進行搜尋，這同時意味著make不會在使用它之後自動刪除它；參閱隱含規則鏈。

對於舊版的make程式，透過一個請求命令，如‘make depend’，利用編譯器的特點生成先決條件是傳統的習慣。這些命令將產生一個‘depend’檔案，該檔案引入(include)所有自動生成的先決條件；然後makefile檔案可以使用include命令將它們讀入（參閱引入(include)其它makefile檔案）。

在GNU make中，重新構造makefile檔案的特點使這個慣例成爲了舊式的東西——您永遠不必具體告訴make重新生成先決條件，因爲GNU make總是重新構造任何舊式的makefile檔案。參閱Makefile檔案的重新生成的過程。

我們推薦使用自動生成先決條件的習慣是把makefile檔案和源程式檔案一一對應起來。如，對每一個源程式檔案‘name.c’有一名爲‘name.d’的makefile檔案和它對應，該makefile檔案中列出了名爲‘name.o’的OBJ檔案所先決條件

的檔案。這種模式的優點是僅在源程式檔案改變的情況下才有必要重新掃描生成新的先決條件。

這裡有一個根據C語言源程式‘name.c’生成名爲‘name.d’先決條件檔案的樣式規則：

```
%.d: %.c
        set -e; $(CC) -M $(CPPFLAGS) $< \
              | sed 's/\($*\)\.o[ :]*/\1.o $@ : /g' > $@; \
             [ -s $@ ] || rm -f $@
```

關於定義樣式規則的訊息參閱定義與重新定義樣式規則。‘-e’開關是告訴shell如果$(CC)命令執行失敗（非零狀態退出）立即退出。正常情況下，shell退出時帶有最後一個命令在管道中的狀態（sed）,因此make不能注意到編譯器產生的非零狀態。

對於GNU C編譯器您可以使用‘-MM’開關代替‘-M’，這是省略了有關係統頭檔案的先決條件。詳細內容參閱《GNU CC使用手冊》中控制預處理選項。
命令Sed的作用是翻譯

（例如）：
main.o : main.c defs.h
到：
main.o main.d : main.c defs.h
這使每一個‘.d’檔案和與之對應的‘.o’檔案依靠相同的源程式檔案和頭檔案，據此，Make可以知道如果任一個源程式檔案和頭檔案發生變化，則必須重新構造先決條件檔案。
一旦您定義了重新構造‘.d’檔案的規則，您可以使用使用include命令直接將它們讀入，（參閱引入(include)其它makefile檔案），

例如：
sources = foo.c bar.c
include $(sources:.c=.d)
（這個例子中使用一個代替變數參照從源程式檔案清單‘foo.c bar.c'翻譯到先決條件檔案清單‘foo.d bar.d'。詳細內容參閱替換引用。）所以，‘.d’的makefile檔案和其它makefile檔案一樣，即使沒用您的任何進一步的指令，make同樣會在必要的時候重新構建它們。參閱Makefile檔案的重新生成過程。

5在規則中使用命令

規則中的命令由一系列shell命令行組成，它們一條一條的按順序執行。除第一條命令行可以分號爲開始附屬在目標-先決條件行後面外，所有的命令行必須以TAB開始。空白行與註釋行可在命令行中間出現，處理時它們被忽略。（但是必須注意，以TAB開始的‘空白行’不是空白行，它是空命令，參閱使用空命令。）
用戶使用多種不同的shell程式，如果在makefile檔案中沒有指明其它的shell，則使用預設的‘/bin/sh’解釋makefile檔案中的命令。參閱命令執行。
使用的shell種類決定了是否能夠在命令行上寫註釋以及編寫註釋使用的語法。當使用‘/bin/sh’作爲shell，以‘#’開始的註釋一直延伸到該行結束。‘#’不必在行首，而且‘#’不是註釋的一部分。

5.1 命令回顯

正常情況下make在執行命令之前首先列印命令行，我們因這樣可將您編寫的命令原樣輸出故稱此爲回顯。
以‘@’起始的行不能回顯，‘@’在傳輸給shell時被丟棄。典型的情況，您可以在makefile檔案中使用一個僅僅用於列印某些內容的命令，如echo命令來顯示makefile檔案執行的進程：

```
@echo About to make distribution files
```

當使用make時給出‘-n’或‘--just-print’標誌，則僅僅回顯命令而不執行命令。參閱選項概要。在這種情況下也只有在這種情況下，所有的命令行都回顯，即使以‘@’開始的命令行也回顯。這個標誌對於在不使用命令的情況下發現make認爲哪些是必要的命令非常有用。
‘-s’或‘--silent’標誌可以使make阻止所有命令回顯，好像所有的行都以‘@’開始一樣。在makefile檔案中使用不帶先決條件的特別目標‘.SILENT’的規則可以達到相同的效果（參閱內建的特殊目標名）。因爲‘@’使用更加靈活以至於現下已基本不再使用特別目標.SILENT。

5.2執行命令

需要執行命令更新目標時，每一命令行都會使用一個獨立的子shell環境，保證該命令行得到執行。（實際上，make可能走不影響結果的捷徑(stem)。）
請注意：這意味著設定局部變數的shell命令如cd等將不影響緊跟著的命令行；如果您需要使用cd命令影響到下一個命令，請把這兩個命令放到一行，它們中間用分號隔開，這樣make將認爲它們是一個單一的命令行，把它們放到一起傳遞給shell，然後按順序執行它們。例如：

```
foo : bar/lose
        cd bar; gobble lose > ../foo
```

如果您喜歡將一個單一的命令分割成多個文字(text)行，您必須用反斜線(\)作爲每一行的結束，最後一行除外。這樣，多個文字(text)行透過刪除反斜線(\)按順序組成一新行，然後將它傳遞給shell。如此，下面的例子和前面的例子是等同的：

```
foo : bar/lose
        cd bar;  \
        gobble lose > ../foo
```

用作shell的程式是由變數SHELL指定，預設情況下，使用程式‘/bin/sh’作爲shell。
在MS_DOS上執行，如果變數SHELL沒有指定，變數COMSPEC的值用來代替指定shell。
在MS_DOS上執行和在其它系統上執行，對於makefile檔案中設定變數SHELL的行的處理也不一樣。因爲MS_DOS的shell，‘command.com’，功能十分有限，所以許多make用戶傾向於安裝一個代替的shell。因此，在MS_DOS上執行，make檢測變數SHELL的值，並根據它指定的Unix風格或DOS風格的shell變化它的行爲。即使使用變數SHELL指向‘command.com’，make依然檢測變數SHELL的值。
如果變數SHELL指定Unix風格的shell，在MS_DOS上執行的make將附加檢查指定的shell是否能真正找到；如果不能找到，則忽略指定的shell。在MS_DOS上，GNU make按照下述步驟搜尋shell:
1、在變數SHELL指定的目錄中。例如，如果makefile指明`SHELL = /bin/sh'，make將在當前路徑(stem)下尋找子目錄‘/bin’。
2、在當前路徑(stem)下。
3、按順序搜尋變數PATH指定的目錄。

在所有搜尋的目錄中，make首先尋找指定的檔案（如例子中的‘sh’）。如果該檔案沒有存在，make將在上述目錄中搜

尋帶有確定的可執行檔案展開的檔案。例如：‘.exe', ‘.com', ‘.bat', ‘.btm', ‘.sh'檔案和其它檔案等。
如果上述過程中能夠成功搜尋一個shell，則變數SHELL的值將設定為所發現shell的全路徑(stem)檔案名。然而如果上述努力全部失敗，變數SHELL的值將不改變，設定shell的行的有效性將被忽略。這是在make執行的系統中如果確實安裝了Unix風格的shell，make僅支援指明的Unix風格shell特點的原因。
注意這種對shell的展開搜尋僅僅限制在makefile檔案中設定變數SHELL的情況。如果在環境或命令行中設定，希望您指定shell的全路徑(stem)檔案名，而且全路徑(stem)檔案名需像在Unix系統中執行的一樣準確無誤。

經過上述的DOS特色的處理，而且您還把‘sh.exe’安裝在變數PATH指定的目錄中，或在makefile檔案內部設定`SHELL = /bin/sh'（和多數Unix的makefile檔案一樣），則在MS_DOS上的執行效果和在Unix上執行完全一樣。
不像其它大多數變數，變數SHELL從不根據環境設定。這是因為環境變數SHELL是用來指定您自己選擇交互使用的shell程式。如果變數SHELL在環境中設定，它將影響makefile檔案的功能，這是非常不劃算的，參閱環境變數。然而在MS-DOS和MS-WINDOWS中在環境中設定變數SHELL的值是要使用的，因為在這些系統中，絕大多數用戶並不設定該變數的值，所以make很可能特意為該變數指定要使用的值。在MS-DOS上，如果變數SHELL的設定對於make不合適，您可以設定變數MAKESHELL用來指定make使用的shell；這種設定將使變數SHELL的值失效。

5.3 並行執行

GNU make可以同時執行幾條命令。正常情況下，make一次執行一個命令，待它完成後在執行下一條命令。然而，使用‘-j’和‘--jobs’選項將告訴make同時執行多條命令。在MS-DOS上，‘-j’選項沒有作用，因為該系統不支援多進程處理。
如果‘-j’選項後面跟一個整數，該整數表示一次執行的命令的條數；這稱為job slots數。如果‘-j’選項後面沒有整數，也就是沒有對job slots的數目限制。預設的job slots數是一，這意味著按順序執行（一次執行一條命令）。同時執行多條命令的一個不太理想的結果是每條命令產生的輸出與每條命令發送的時間對應，即命令產生的消息回顯可能較為混亂。
另一個問題是兩個進程不能使用同一設備輸入，所以必須確定一次只能有一條命令從終端輸入，make只能保證正在執行的命令的標準輸入流有效，其它的標準輸入流將失效。這意味著如果有幾個同時從標準輸入設備輸入的話，對於絕大多數子進程將產生致命的錯誤（即產生一個‘Broken pipe’信號）。

命令對一個有效的標準輸入流（它從終端輸入或您為make改造的標準輸入設備輸入）的需求是不可預測的。第一條執行的命令總是第一個得到標準輸入流，在完成一條命令後第一條啟動的另一條命令將得到下一個標準輸入流，等等。
如果我們找到一個更好替換方案，我們將改變make的這種工作模式。在此期間，如果您使用並行處理的特點，您不應該使用任何需要標準輸入的命令。如果您不使用該特點，任何需要標準輸入的命令將都能正常工作。
最後，make的遞迴也導致出現問題。更詳細的內容參閱與子make通訊的選項。
如果一個命令失敗（被一個信號中止，或非零退出），且該條命令產生的錯誤不能忽略（參閱命令錯誤），剩餘的構建同一目標的命令行將會停止工作。如果一條命令失敗，而且‘-k’或‘--keep-going’選項也沒有給出（參閱選項概要），make將放棄繼續執行。如果make由於某種原因（包括信號）要中止，此時又子進程正在執行，它將等到這些子進程結束之後再實際退出。
當系統正滿負荷執行時，您或許希望在負荷輕的時再添加任務。這時，您可以使用‘-l’選項告訴make根據平均負荷限制同一時刻執行的任務數量。‘-l’或‘--max-load’選項一般後跟一個浮點數。例如：

-l 2.5
將不允許make在平均負荷高於2.5時啟動一項任務。‘-l’選項如果沒有跟數據，則取消前面‘-l’給定的負荷限制。
更精確地講，當make啟動一項任務時，而它此時已經有至少一項任務正在執行，則它將檢查當前的平均負荷；如果不低於‘-l’選項給定的負荷限制時，make將等待直到平均負荷低於限制或所有其它任務完成後再啟動其它任務。
預設情況下沒有負荷限制。

5.4命令錯誤

在每一個shell命令返回後，make檢查該命令退出的狀態。如果該命令成功地完成，下一個命令行就會在新的子shell環境中執行，當最後一個命令行完成後，這條規則也宣告完成。如果出現錯誤（非零退出狀態），make將放棄當前的規則，也許是所有的規則。
有時一個特定的命令失敗並不是出現了問題。例如：您可能使用mkdir命令建立一個目錄存在，如果該目錄已經存在，mkdir將報告錯誤，但您此時也許要make繼續執行。

要忽略一個命令執行產生的錯誤，請使用字符‘-’（在初始化TAB的後面）作爲該命令行的開始。字符‘-’在命令傳遞給shell執行時丟棄。例如：

```
clean:
        -rm -f *.o
```

這條命令即使在不能刪除一個檔案時也強製rm繼續執行。
在執行make時使用‘-i’或‘--ignore-errors’選項，將會忽略所有規則的命令執行產生的錯誤。在makefile檔案中使用如果沒有先決條件的特殊目標.IGNORE規則，也具有同樣的效果。但因爲使用字符‘-’更靈活，所以該條規則已經很少使用。

一旦使用‘-’或‘-i’選項，執行命令時產生的錯誤被忽略，此時make像處理成功執行的命令一樣處理具有返回錯誤的命令，唯一不同的地方是列印一條消息，告訴您命令退出時的編碼狀態，並說明該錯誤已經被忽略。如果發生錯誤而make並不說明其被忽略，則暗示當前的目標不能成功重新構造，並且和它直接相關或間接相關的目標同樣不能重建。因爲前一個過程沒有完成，所以不會進一步執行別的命令。
在上述情況下，make一般立即放棄任務，返回一個非零的狀態。然而，如果指定‘-k’或‘--keep-goning’選項，make則繼續考慮這個目標的其它先決條件，如果有必要在make放棄返回非零狀態之前重建它們。例如，在編譯一個OBJ檔案發生錯誤後，即使make已經知道將所有OBJ檔案連接在一起是不可能的，‘make -k'選項也繼續編譯其它OBJ檔案。詳細內容參閱選項概要。通常情況下，make的行爲基於假設您的目的是更新指定的目標，一旦make得知這是不可能的，它將立即報告失敗。‘-k’選項是告訴make真正的目的是測試程式中所有變化的可行性，或許是尋找幾個獨立的問題以便您可以在下次編譯之前糾正它們。這是Emacs編譯命令預設情況下傳遞‘-k’選項的原因。

通常情況下，當一個命令執行失敗時，如果它已經改變了目標檔案，則該檔案很可能發生混亂而不能使用或該檔案至少沒有完全得到更新。但是，檔案的時間戳卻表明該檔案已經更新到最新，因此在make下次執行時，它將不再更新該檔案。這種狀況和命令被發出的信號強行關閉一樣，參閱中斷或關閉make。因此，如果在開始改變目標檔案後命令出錯，一般應該刪除目標檔案。如果.DELETE_ON_ERROR作爲目標在makefile檔案中出現，make將自動做這些事情。這是您應該明確要求make執行的動作，不是以前的慣例；特別考慮到兼容性問題時，您更應明確提出這樣的要求。

5.5中斷或關閉make

如果make在一條命令執行時得到一個致命的信號， 則make將根據第一次檢查的時間戳和最後更改的時間戳是否發生變化決定它是否刪除該命令要更新的目標檔案。
刪除目標檔案的目的是當make下次執行時確保目標檔案從原檔案得到更新。爲什麼？假設正在編譯檔案時您鍵入Ctrl-c，而且這時已經開始寫OBJ檔案‘foo.o’，Ctrl-c關閉了該編譯器，結果得到不完整的OBJ檔案‘foo.o’的時間戳比源程式‘foo.c’的時間戳新，如果make收到Ctrl-c的信號而沒有刪除OBJ檔案‘foo.o’，下次請求make更新OBJ檔案‘foo.o’時，make將認爲該檔案已更新到最新而沒有必要更新，結果在linker將OBJ檔案連接爲可執行檔案時產生奇怪的錯誤訊息。

您可以將目標檔案作爲特殊目標.PRECIOUS的先決條件從而阻止make這樣刪除該目標檔案。在重建一個目標之前，make首先檢查該目標檔案是否出現下特殊目標.PRECIOUS的先決條件清單中，從而決定在信號發生時是否刪除該目標檔案。您不刪除這種目標檔案的原因可能是：目標更新是一種原子風格，或目標檔案存在僅僅爲了記錄更改時間（其內容無關緊要），或目標檔案必須一直存在，用來防止其它類型的錯誤等。

5.6遞迴make

遞迴意味著可以在makefile檔案中將make作爲一個命令使用。這種技術在引入(include)大的系統中把makefile分離爲各種各樣的子系統時非常有用。例如，假設您有一個子目錄‘subdir’，該目錄中有它自己的makefile檔案，您希望在該子目錄中執行make時使用該makefile檔案，則您可以按下述模式編寫：

```
  subsystem:
                cd subdir && $(MAKE)
```

或, 等同於這樣寫 (參閱選項概要):

```
subsystem:
        $(MAKE) -C subdir
```

您可以僅僅拷貝上述例子實現make的遞迴，但您應該了解它們是如何工作的，它們爲什麼這樣工作，以及子make和上層make的相互關係。

爲了使用方便，GNU make把變數CURDIR的值設定爲當前工作的路徑(stem)。如果‘-C’選項有效，它將引入(include)的是新路徑(stem)，而不是原來的路徑(stem)。該值和它在makefile中設定的值有相同的優先權（預設情況下，環境變數CURDIR不能重載）。注意，操作make時設定該值無效。

5.6.1 變數MAKE的工作模式

遞迴make的命令總是使用變數MAKE，而不是明確的命令名‘make’，如下所示：

```
subsystem:
        cd subdir && $(MAKE)
```

該變數的值是呼叫make的檔案名。如果這個檔案名是‘/bin/make’，則執行的命令是`cd subdir && /bin/make'。如果您在上層makefile檔案時用特定版本的make，則執行遞迴時也使用相同的版本。
在命令行中使用變數MAKE可以改變‘-t' (‘--touch'), ‘-n' (‘--just-print'), 或 ‘-q' (‘--question')選項的效果。如果在使用變數MAKE的命令行首使用字符‘+’也會起到相同的作用。參閱代替執行命令。
設想一下在上述例子中命令‘make -t’的執行過程。（‘-t’選項標誌目標已經更新，但卻不執行任何命令，參閱代替執行命令。）按照通常的定義，命令‘make t’在上例中僅僅建立名爲‘subsystem’的檔案而不進行別的工作。您實際要求執行‘cd subdir && make t’干什麼？是執行命令或是按照‘-t’的要求不執行命令？
Make的這個特點是這樣的：只要命令行中引入(include)變數MAKE，標誌`-t', `-n' 和 `-q'將不對本行起作用。雖然存在標誌不讓命令執行，但引入(include)變數MAKE的命令行卻正常執行，make實際上是透過變數MAKEFLAGS將標誌值傳遞給了子make（參閱與子make通訊的選項）。所以您的驗証檔案、列印命令的請求等都能傳遞給子系統。

5.6.2與子make通訊的變數

透過明確要求，上層make變數的值可以借助環境傳遞給子make，這些變數能在子make中預設定義，在您不使用‘-e’開關的情況下，傳遞的變數的值不能代替子make使用的makefile檔案中指定的值（參閱命令概要）。
向下傳遞、或輸出一個變數時，make將該變數以及它的值添加到執行每一條命令的環境中。子make，作爲附應，使用該環境初始化它的變數值表。參閱環境變數。

除了明確指定外，make僅向下輸出在環境中定義並初始化的或在命令行中設定的變數，而且這些變數的變數名必須僅由字母、數字和下劃線組成。一些shell不能處理名字中含有字母、數字和下劃線以外字符的環境變數。特殊變數如SHELL和MAKEFLAGS一般總要向下輸出（除非您不輸出它們）。即使您把變數MAKEFILE設爲其它的值，它也向下輸出。
Make自動傳遞在命令行中定義的變數的值，其方法是將它們放入MAKEFLAGS變數中。詳細內容參閱下節。Make預設創造的變數的值不能向下傳遞，子make可以自己定義它們。如果您要將指定變數輸出給子make，請用export指令，格式如下：

```
export variable ...
```

您要將阻止一些變數輸出給子make，請用unexport指令，格式如下：

```
unexport variable ...
```

爲方便起見，您可以同時定義並輸出一個變數：

```
export variable = value
```

下面的格式具有相同的效果:

variable = value
export variable

以及

export variable := value

具有相同的效果：

variable := value
export variable

同樣,

export variable += value

亦同樣:

variable += value

export variable
參閱為變數值附加文字(text)。
您可能注意到export和unexport指令在make與shell中的工作模式相同，如sh。
如果您要將所有的變數都輸出，您可以單獨使用export：
export
這告訴make 把export和unexport沒有提及的變數統統輸出，但任何在unexport提及的變數仍然不能輸出。如果您單獨使用export作為預設的輸出變數模式，名字中含有字母、數字和下劃線以外字符的變數將不能輸出，這些變數除非您明確使用export指令提及才能輸出。

單獨使用export的行為是頭家本GNU make預設定義的行為。如果您的makefile依靠這些行為，而且您希望和頭家本GNU make兼容，您可以為特殊目標.EXPORT_ALL_VARIABLES 編寫一條規則代替export指令，它將被頭家本GNU make忽略，但如果同時使用export指令則報錯。

同樣，您可以單獨使用unexport告訴make預設不要輸出變數，因為這是預設的行為，只有前面單獨使用了export（也許在一個包括的makefile中）您才有必要這樣做。您不能同時單獨使用export和unexport指令實現對某些命令輸出對其它的命令不輸出。最後面的一條指令（export或unexport）將決定make的全部執行結果。

作為一個特點，變數MAKELEVEL的值在從一個層次向下層傳遞時發生變化。該變數的值是字符型，它用十進製數表示層的深度。'0'代表頂層make，'1'代表子make，'2'代表子-子-make，以此類推。Make為一個命令建立一次環境，該值增加1。

該變數的主要作用是在一個條件指令中測試（參閱makefile檔案的條件語句）；採用這種方法，您可以編寫一個makefile，如果遞迴採用一種執行模式，由您控制直接執行採用另一種執行模式。
您可以使用變數MAKEFILES使所有的子make使用附加的makefile檔案。變數MAKEFILES的值是makefile檔案名的清單，檔案名之間用空格隔開。在外層makefile中定義該變數，該變數的值將透過環境向下傳遞；因此它可以作為子make的額外的makefile檔案，在子make讀正常的或指定的makefile檔案前，將它們讀入。參閱變數MAKEFILES。

5.6.3與子make通訊的選項

諸如'-s'和'-k'標誌透過變數MAKEFLAGS自動傳遞給子make。該變數由make自動建立，並引入(include)make收到的標誌字母。所以，如果您是用'make ks'變數MAKEFLAGS就得到值'ks'。
作為結果，任一個子make都在它的執行環境中為變數MAKEFLAGS賦值；作為附應，make使用該值作為標誌並進行處理，就像它們作為參數被給出一樣。參閱選項概要。

同樣，在命令行中定義的變數也將借助變數MAKEFLAGS傳遞給子make。變數MAKEFLAGS值中的字可以引入(include)'='，make將它們按變數定義處理，其過程和在命令行中定義的變數一樣。參閱變數重載。
選項`-C', `-f', `-o', 和 '-W'不能放入變數MAKEFLAGS中；這些選項不能向下傳遞。

'-j'選項是一個特殊的例子（參閱並行執行）。如果您將它設定為一些數值'N'，而且您的作業系統支援它（大多數Unix系統支援，其它作業系統不支援），父make和所有子make通訊保證在它們中間同時僅有'N'個任務執行。注意，任何引入(include)遞迴的任務（參閱代替執行命令）不能計算在總任務數內（否則，我們僅能得到'N'個子make執行，而沒有多餘的時間片執行實在的工作）。

如果您的作業系統不支援上述通訊機製，那麼'-j 1'將放到變數MAKEFLAGS中代替您指定的值。這是因為如果'-j'選項傳遞給子make，您可能得到比您要求多很多的並行執行的任務數。如果您給出'-j'選項而沒有數字參數，意味著儘可能並行處理多個任務，這樣向下傳遞，因為倍數的無限制性所以至多為1。
如果您不希望其它的標誌向下傳遞，您必須改變變數MAKEFLAGS的值，其改變模式如下：

```
subsystem:
        cd subdir && $(MAKE) MAKEFLAGS=
```

該命令行中定義變數的實際上出現下變數MAKEOVERRIDES中，而且變數MAKEFLAGS引入(include)了該變數的引用值。如果您要向下傳遞標誌，而不向下傳遞命令行中定義的變數，這時，您可以將變數MAKEOVERRIDES的值設為空，格式如下：
MAKEOVERRIDES =

這並不十分有用。但是，一些系統對環境的大小有固定限制，而且該值較小，將這么多的訊息放到變數MAKEFLAGS的值中可能超過該限制。如果您看到 'Arg list too long'的錯誤訊息，很可能就是由於該問題造成的。（按照嚴格的POSIX.2的規定，如果在makefile檔案定義特殊目標 '.POSIX' ，改變變數MAKEOVERRIDES的值並不影響變數MAKEFLAGS。也許您並不關心這些。）

爲了和早期版本兼容，具有相同功能的變數MFLAGS也是存在的。除了它不能引入(include)命令行定義變數外，它和變數MAKEFLAGS有相同的值，而且除非它是空值，它的值總是以短線開始（MAKEFLAGS只有在和多字符選項一起使用時才以短線開始，如和 '--warn-undefined-variables' 連用）。變數MFLAGS傳統的使用在明確的遞迴make的命令中，例如：

```
subsystem:
        cd subdir && $(MAKE) $(MFLAGS)
```

但現下，變數MAKEFLAGS使這種用法變得多餘。如果您要您的makefile檔案和老版本的make程式兼容，請使用這種模式；這種模式在現代版本make中也能很好的工作。

如果您要使用每次執行make都要設定的特定選項，例如 '-k' 選項（參閱選項概要），變數MAKEFLAGS十分有用。您可以簡單的在環境中將給變數MAKEFLAGS賦值，或在makefile檔案中設定變數MAKEFLAGS，指定的附加標誌可以對整個makefile檔案都起作用。（注意：您不能以這種模式使用變數MFLAGS，變數MFLAGS存在僅爲和早期版本兼容，採用其它模式設定該變數make將不予解釋。）

當make解釋變數MAKEFLAGS值的時候（不管在環境中定義或在makefile檔案中定義），如果該值不以短線開始，則make首先爲該值假設一個短線；接著將該值分割成字，字與字間用空格隔開，然後將這些字進行語法分析，好像它們是在命令行中給出的選項一樣。（ '-C', '-f', '-h', '-o', '-W'選項以及它們的長名字版本都將忽略，對於無效的選項不產生錯誤訊息。）

如果您在環境中定義變數MAKEFLAGS，您不要使用嚴重影響make執行，破壞makefile檔案的意圖以及make自身的選項。例如 '-t', '-n', 和 '-q'選項，如果將它們中的一個放到變數MAKEFLAGS的值中，可能產生災難性的後果，或至少產生讓人討厭的結果。

5.6.4 '--print-directory' 選項

如果您使用幾層make遞迴，使用 '-w' 或 '--print-directory' 選項，透過顯示每個make開始處理以及處理完成的目錄使您得到比較容易理解的輸出。例如，如果使用 'make w' 命令在目錄 '/u/gnu/make' 中執行make，則make將下面格式輸出訊息：

make: Entering directory `/u/gnu/make'.

說明進入目錄中，還沒有進行任何任務。下面的訊息：

make: Leaving directory `/u/gnu/make'.

說明任務已經完成。

通常情況下，您不必具體指明這個選項，因爲make已經爲您做了：當您使用 '-C' 選項時， '-w' 選項已經自動打開，在子make中也是如此。如果您使用 '-s' 選項， '-w' 選項不會自動打開，因爲 '-s' 選項是不列印訊息，同樣使用`--no-print-directory'選項 '-w' 選項也不會自動打開。

5.7定義固定次序命令

在建立各種目標時，相同次序的命令十分有用時，您可以使用define指令定義固定次序的命令，並根據這些目標的規則引用固定次序。固定次序實際是一個變數，因此它的名字不能和其它的變數名衝突。

下面是定義固定次序命令的例子：

```
define run-yacc
yacc $(firstword $^)
mv y.tab.c $@
endef
```

run-yacc是定義的變數的名字；endef標誌定義結束；中間的行是命令。define指令在固定次序中不對變數引用和函數呼叫展開；字符 '$' 、圓括號、變數名等等都變成您定義的變數的值的一部分。***定義多行變數一節對指令define有詳細解釋。***

在該例子中，對於任何使用該固定次序的規則，第一個命令是對其第一個先決條件執行Yacc命令，Yacc命令執行產生的輸出檔案一律命名爲 'y.tab.c' ；第二條命令，是將該輸出檔案的內容移入規則的目標檔案中。

在使用固定次序時，規則中命令使用的變數應被確定的值替代，您可以像替代其它變數一樣替代這些變數（詳細內容參閱變數引用基礎）。因爲由define指令定義的變數是遞歸展開的變數，所以在使用時所有變數引用才展開。例如：

foo.c : foo.y

```
        $(run-yacc)
```

當固定次序‘run-yacc’執行時，‘foo.y’將代替變數‘$^’，‘foo.c’將代替變數‘$@’。

這是一個現實的例子，但並不是必要的，因爲make有一條隱含規則可以根據涉及的檔案名的類型確定所用的命令。參閱使用隱含規則。

在命令執行時，固定次序中的每一行被處理爲和直接出現下規則中的命令行一樣，前面加上一個Tab，make也特別爲每一行請求一個獨立的子shell。您也可以在固定次序的每一行上使用影響命令行的前綴字符(`@', `-',和 `+')，參閱在規則中使用命令。例如使用下述的固定次序：

```
@echo "frobnicating target $@"
frob-step-1 $< -o $@-step-1
frob-step-2 $@-step-1 -o $@
endef
```

make將不回顯第一行，但要回顯後面的兩個命令行。

另一方面，如果前綴字符在引用固定次序的命令行中使用，則該前綴字符將應用到固定次序的每以行中。例如這個規則：

```
frob.out: frob.in
        @$(frobnicate)
```

將不回顯固定次序的任何命令。具體內容參閱命令回顯。

5.8 使用空命令

定義什麼也不干的命令有時很有用，定義空命令可以簡單的給出一個僅僅含有空格而不含其它任何東西的命令即可。例如：

```
target: ;
```

爲字元串‘target’定義了一個空命令。您也可以使用以Tab字符開始的命令行定義一個空命令，但這由於看起來空白容易造成混亂。

也許您感到奇怪，爲什麼我們定義一個空命令？唯一的原因是爲了阻止目標更新時使用隱含規則提供的命令。（參閱使用隱含規則以及定義最新類型的預設規則）也許您喜愛爲實際不存在的目標檔案定義空命令，因爲這樣它的先決條件可以重建。然而這樣做並不是一個好方法，因爲如果目標檔案實際存在，則先決條件有可能不重建，使用假想(phony)目標是較好的選擇，參閱假想(phony)目標。

6 使用變數

變數是在makefile中定義的名字，其用來代替一個文字(text)字元串，該文字(text)字元串稱為該變數的值。在具體要求下，這些值可以代替目標、先決條件、命令以及makefile檔案中其它部分。（在其它版本的make中，變數稱為巨集（macros）。）
在makefile檔案讀入時，除規則中的shell命令、使用'='定義的'='右邊的變數、以及使用define指令定義的變數體此時不展開外，makefile檔案其它各個部分的變數和函數都將展開。
變數可以代替檔案清單、傳遞給編譯器的選項、要執行的程式、查找源檔案的目錄、輸出寫入的目錄，或您可以想像的任何文字(text)。

變數名是不包括 **':', '#', '='、前導或結尾空格的任何字元串**。然而變數名包含字母、數字以及下劃線以外的其它字符的情況應盡量避免，因為它們可能在將來被賦予特別的含義，而且對於一些shell它們也不能透過環境傳遞給子make（參閱與子make通訊的變數）。**變數名是大小寫敏感的**，例如變數名'foo', 'FOO', 和 'Foo'代表不同的變數。
使用大寫字母作為變數名是以前的習慣，但我們推薦在makefile內部使用小寫字母作為變數名，預留大寫字母作為控制隱含規則**參數或用戶重載命令選項參數的變數名**。參閱變數重載。
一少部分的變數使用一個標點符號或幾個字符作為變數名，這些變數是自動變數，它們又特定的用途。參閱自動變數。

6.1 變數引用基礎

寫一個美元符號後跟用圓括號或大括號括住變數名則可引用變數的值：'$(foo)' 和 '${foo}'都是對變數'foo'的有效引用。'$'的這種特殊作用是您在命令或檔案名中必須寫'$$'才有單個'$'的效果的原因。
變數的引用可以用在上下文的任何地方：目標、先決條件、命令、絕大多數指令以及新變數的值等等。這裡有一個常見的例子，在程式中，變數儲存著所有OBJ檔案的檔案名：

```
objects = program.o foo.o utils.o
program : $(objects)
        cc -o program $(objects)

$(objects) : defs.h
```

變數的引用按照嚴格的文字(text)替換進行，這樣該規則

```
foo = c
prog.o : prog.$(foo)
        $(foo)$(foo) -$(foo) prog.$(foo)
```

可以用於編譯C語言源程式'prog.c'。因為在變數分發時，變數值前面的空格被忽略，所以變數foo的值是'C'。（不要在您的makefile檔案這樣寫！）
美元符號後面跟一個字符但不是美元符號、圓括號、大括號，則該字符將被處理為單字符的變數名。因此可以使用'$x'引用變數x。然而，這除了在使用自動變數的情況下，在其它實際工作中應該完全避免。參閱自動變數。

6.2 變數的兩個特色

在GNU make中可以使用兩種模式為變數賦值，我們將這兩種模式稱為變數的兩個特色（two flavors）。兩個特色的區別在於它們的定義模式和展開時的模式不同。
變數的第一個特色是遞迴展開型變數。這種類型的變數定義模式：在命令行中使用 **'='** 定義（參閱設定變數）或使用**define**指令定義（參閱定義多行變數）。變數替換對於您所指定的值是逐字進行替換的；如果它包含對其它變數的引用，這些引用在該變數替換時（或在展開為其它字元串的過程中）才被展開。這種展開模式稱為遞迴型展開。例如：

```
foo = $(bar)
bar = $(ugh)
ugh = Huh?

all:;echo $(foo)
```

將回顯'Huh?'：'$(foo)'展開為'$(bar)'，進一步展開為'$(ugh)'，最終展開為'Huh?'。
這種特色的變數是其它版本make支援的變數類型，有缺點也有優點。大多數人認為的該類型的變數的優點是：

```
CFLAGS = $(include_dirs) -O
include_dirs = -Ifoo -Ibar
```

即能夠完成希望它完成的任務：當‘CFLAGS’在命令中展開時，它將最終展開爲‘-Ifoo -Ibar’。其最大的缺點是不能在變數後追加內容，如在：
CFLAGS = $(CFLAGS) -O

在變數展開過程中可能導致無窮循環（實際上make偵測到無窮循環就會產生錯誤訊息）。
它的另一個缺點是在定義中引用的任何函數時（參閱文字(text)轉換函數）變數一旦展開函數就會立即執行。這可導致make執行變慢，性能變壞；並且導致萬用字元與shell函數（因不能控制何時呼叫或呼叫多少次）產生不可預測的結果。爲避免該問題和遞迴展開型變數的不方便性，出現了另一個特色變數：簡單展開型變數。
簡單展開型變數在命令行中用**‘:=’**定義（參閱設定變數）。簡單展開型變數的值是一次掃描永遠使用，對於引用的其它變數和函數在定義的時候就已經展開。簡單展開型變數的值實際就是您寫的文字(text)展開的結果。因此它不包含任何對其它變數的引用；在該變數定義時就包含了它們的值。所以：

```
x := foo
y := $(x) bar
x := later
```
等同於：
```
y := foo bar
x := later
```

引用一個簡單展開型變數時，它的值也是逐字替換的。這裡有一個稍複雜的例子，說明了‘:=’和shell函數連接用法（參閱函數shell）。該例子也表明了變數MAKELEVEL的用法，該變數在層與層之間傳遞時值發生變化。（參閱與子make通訊的變數，可獲得變數MAKELEVEL關於的訊息。）

```
ifeq (0,${MAKELEVEL})
cur-dir   := $(shell pwd)
whoami    := $(shell whoami)
host-type := $(shell arch)
MAKE := ${MAKE} host-type=${host-type} whoami=${whoami}
endif
```

按照這種方法使用‘:=’的優點是看起來像下述的典型的‘下降到目錄’的命令：
```
${subdirs}:
    ${MAKE} cur-dir=${cur-dir}/$@ -C $@ all
```

簡單展開型變數因爲在絕大多數程式設計語言中可以像變數一樣工作，因此它能夠使複雜的makefile程式更具有預測性。它們允許您使用它自己的值重新定義（或它的值可以被一個展開函數以某些模式處理），它們還允許您使用更有效的展開函數（參閱文字(text)轉換函數）。
您可以使用簡單展開型變數將控制的前導空格引入到變數的值中。前導空格字符一般在變數引用和函數呼叫時被丟棄。簡單展開型變數的這個特點意味著您可以在一個變數的值中包含前導空格，並在變數引用時保護它們。像這樣：
```
nullstring :=
space := $(nullstring) # end of the line
```
這裡變數space的值就是一個空格，註釋‘# end of the line’包括在這裡爲了讓人更易理解。因爲尾部的空格不能從變數值中分離出去，僅在結尾留一個空格也有同樣的效果（但是此時相當難讀），如果您在變數值後留一個空格，像這樣在行的結尾寫上註釋清楚表明您的打算是很不錯的主意。相反，如果您在變數值後不要空格，您千萬記住不要在行的後面留下幾個空格再隨意放入註釋。
例如：
```
dir := /foo/bar    # directory to put the frobs in
```
這裡變數dir的值是‘/foo/bar ’（四個尾部空格），這不是預期的結果。（假設‘/foo/bar’是預期的值）。
另一個給變數賦值的操作符是**‘?=’**，它稱爲**條件變數賦值操作符**，因爲它僅僅在變數還沒有定義的情況下有效。這聲明：
```
FOO ?= bar
```
和下面的語句嚴格等同（參閱函數origin）
```
ifeq ($(origin FOO), undefined)
  FOO = bar
```

endif
注意，一個變數即使是空值，它仍然已被定義，所以使用‘?=’定義無效。

6.3變數引用進階技術

本節內容介紹變數引用的進階技術。

6.3.1替換引用

替**換引用是用您指定的變數替換一個變數的值。它的形式**‘$(var:a=b)’（或‘${var:a=b}’），它的含義是把變數var的值中的每一個字結尾的a用b替換。
我們說‘在一個字的結尾’，我們的意思是a一定在一個字的結尾出現，且a的後面要么是空格要么是該變數值的結束，這時的a被替換，值中其它地方的a不被替換。例如：

```
foo := a.o b.o c.o
bar := $(foo:.o=.c)
```

將變數‘bar’的值設爲‘a.c b.c c.c’。參閱變數設定。

替換引用實際是使用展開函數patsubst的簡寫形式（參閱字元串替換和分析函數）。我們提供替換引用也是使展開函數patsubst與make的其它實現手段兼容的措施。
另一種替換引用是使用強大的展開函數patsubst。它的形式和上述的‘$(var:a=b)’一樣，不同在於它必須包含單個‘**%**’字符，其實這種形式等同於‘$(patsubst a,b,$(var))’。有關於函數patsubst展開的描述參閱字元串替換和分析函數。例如：

```
foo := a.o b.o c.o
bar := $(foo:%.o=%.c)
```

社値變數‘bar'的值爲‘a.c b.c c.c'。

6.3.2巢狀變數引用（計算的變數名）

巢狀變數引用（計算的變數名）是一個複雜的概念，僅僅在十分複雜的makefile程式中使用。絕大多數情況您不必考慮它們，僅僅知道建立名字中含有美元標誌的變數可能有奇特的結果就足夠了。然而，如果您是要把一切搞明白的人或您實在對它們如何工作有興趣，請認真閱讀以下內容。
變數可以在它的名字中引用其它變數，這稱爲巢狀變數引用（計算的變數名）。例如：

```
x = y
y = z
a := $($(x))
```

定義阿a爲‘z’：‘$(x)’在‘$($(x))’中展開爲‘y’，因此‘$($(x))’展開爲‘$(y)’，最終展開爲‘z’。這裡對引用的變數名的陳述不太明確；它根據‘$(x)’的展開進行計算，所以引用‘$(x)’是巢狀在外層變數引用中的。
前一個例子表明了兩層巢狀，但是任何層次數目的巢狀都是允許的，例如，這裡有一個三層巢狀的例子：

```
x = y
y = z
z = u
a := $($($(x)))
```

這裡最裡面的‘$(x)’展開爲‘y’，因此‘$($(x))’展開爲‘$(y)’，‘$(y)’展開爲‘z’，最終展開爲‘u’。
在一個變數名中引用遞迴展開型變數，則按通常的風格再展開。例如：

```
x = $(y)
y = z
z = Hello
a := $($(x))
```

定義的a是‘Hello’：‘$($(x))’展開爲‘$($(y))’，‘$($(y))’變爲‘$(z)’, $(z)’最終展開爲‘Hello’。
巢狀變數引用和其它引用一樣也可以包含修改引用和函數呼叫（參閱文字(text)轉換函數）。例如，使用函數subst（參閱字元串替換和分析函數）：

```
x = variable1
variable2 := Hello
```

```
y = $(subst 1,2,$(x))
z = y
a := $($($(z)))
```
定義的a是‘Hello’ 。任何人也不會寫像這樣令人費解的巢狀引用程式，但它確實可以工作：‘$($($(z)))’ 展開為‘$($(y))’ ，‘$($(y))’變為‘$(subst 1,2,$(x))’ 。它從變數‘x’得到值‘variable1’,變換替換為‘variable2’ ，所以整個字元串變為‘$(variable2)’ ，一個簡單的變數引用，它的值為‘Hello’ 。
巢狀變數引用不都是簡單的變數引用，它可以包含好幾個變數引用，同樣也可包含一些固定文字(text)。例如，

```
a_dirs := dira dirb
1_dirs := dir1 dir2

a_files := filea fileb
1_files := file1 file2

ifeq "$(use_a)" "yes"
a1 := a
else
a1 := 1
endif

ifeq "$(use_dirs)" "yes"
df := dirs
else
df := files
endif

dirs := $($(a1)_$(df))
```
根據設定的use_a和use_dirs的輸入可以將dirs這個相同的值分別賦給a_dirs, 1_dirs, a_files 或 1_files。

巢狀變數引用也可以用於替換引用：
```
a_objects := a.o b.o c.o
1_objects := 1.o 2.o 3.o

sources := $($(a1)_objects:.o=.c)
```
根據a1的值，定義的sources可以是`a.c b.c c.c' 或`1.c 2.c 3.c'。
使用巢狀變數引用唯一的限制是它們不能只部分指定要呼叫的函數名，這是因為用於識別函數名的測試在巢狀變數引用展開之前完成。例如：
```
ifdef do_sort
func := sort
else
func := strip
endif

bar := a d b g q c

foo := $($(func) $(bar))
```
則給變數‘foo’的值賦為‘sort a d b g q c' 或 ‘strip a d b g q c'，而不是將‘a d b g q c’作為函數sort或strip的參數。如果在將來去掉這種限制是一個不錯的主意。

您也可以變數賦值的左邊使用巢狀變數引用，或在define指令中。如：
```
dir = foo
$(dir)_sources := $(wildcard $(dir)/*.c)
define $(dir)_print
```

lpr $($(dir)_sources)
endef
該例子定義了變數‘dir', ‘foo_sources', 和 ‘foo_print'。
注意：雖然巢狀變數引用和遞迴展開型變數都是用在複雜的makefile檔案中，但二者不同（參閱變數的兩個特色）。

6.4變數取值

- 變數有以下幾種模式取得它們的值：
- 您可以在執行make時爲變數指定一個重載值。參閱變數重載。
- 您可以在makefile文件中指定值，即變數賦值（參閱***設置變數***）或使用逐字定義變數（參閱***定義多行變數***）。
- 把環境變數變爲make的變數。參閱環境變數。
- 自動變數可根據規則提供值，它們都有簡單的習慣用法，參閱***自動變數***。

變數可以用常量初始化。參閱隱含規則使用的變數。

6.5設定變數

在makefile檔案中設定變數，編寫以變數名開始後跟‘=’ 或 ‘:=’ 的一行即可。任何跟在‘=’ 或 ‘:=’ 後面的內容就變爲變數的值。例如：
objects = main.o foo.o bar.o utils.o
定義一個名爲objects的變數，變數名前後的空格和緊跟‘=’的空格將被忽略。
使用‘=’定義的變數是遞迴展開型變數；以‘:=’定義的變數是簡單展開型變數。簡單展開型變數定義可以包含變數引用，而且變數引用在定義的同時就被立即展開。參閱變數的兩種特色。
變數名中也可以包含變數引用和函數呼叫，它們在該行讀入時展開，這樣可以計算出能夠實際使用的變數名。
變數值的長度沒有限制，但受限於計算機中的實際交換空間。當定義一個長變數時，在合適的地方插入反斜線(\)，把變數值分爲多個文字(text)行是不錯的選擇。這不影響make的功能，但可使makefile檔案更加易讀。
絕大多數變數如果您不爲它設定值，空字元串將自動作爲它的初值。雖然一些變數有內建的非空的初始化值，但您可隨時按照通常的模式爲它們賦值（參閱隱含規則使用的變數。）另外一些變數可根據規則自動設定新值，它們被稱爲自動變數。參閱自動變數。

如果您喜歡僅對沒有定義過的變數賦給值，您可以使用速記符‘?=’代替‘=’。下面兩種設定變數的模式完全等同（參閱函數origin）：
FOO ?= bar
和
ifeq ($(origin FOO), undefined)
FOO = bar
endif

6.6 爲變數值附加文字(text)

爲已經定以過的變數的值追加更多的文字(text)一般比較有用。您可以在獨立行中使用‘+=’來實現上述設想。如：
objects += another.o
這爲變數objects的值添加了文字(text)‘another.o’（其前面有一個前導空格）。這樣：
objects = main.o foo.o bar.o utils.o
objects += another.o
變數objects 設定爲‘main.o foo.o bar.o utils.o another.o'。
使用 `+=' 相同於:
objects = main.o foo.o bar.o utils.o
objects := $(objects) another.o

對於使用複雜的變數值，不同方法的差別非常重要。如變數在以前沒有定義過，則‘+=’的作用和‘=’相同：它定義一個遞迴型變數。然而如果在以前有定義，‘+=’的作用先決條件於您原始定義的變數的特色，詳細內容參閱變數的兩種特色。
當您使用‘+=’爲變數值附加文字(text)時，make的作用就好像您在初始定義變數時就引入(include)了您要追加的文字(text)。如果開始您使用‘:=’定義一個簡單展開型變數，再用‘+=’對該簡單展開型變數值附加文字(text)，則該變數按新的文字(text)值展開，好像在原始定義時就將附加文字(text)定義上一樣，詳細內容參閱設定變數。實際上，

variable := value
variable += more
等同於：
variable := value
variable := $(variable) more

另一方面，當您把‘+=’和首次使用無符號‘=’定義的遞迴型變數一起使用時，make的執行模式會有所差異。在您引用遞迴型變數時，make並不立即在變數引用和函數呼叫時展開您設定的值；而是將它逐字儲存起來，將變數引用和函數呼叫也儲存起來，以備以後展開。當您對於一個遞迴型變數使用‘+=’時，相當於對一個不展開的文字(text)追加新文字(text)。
variable = value
variable += more
粗略等同於：
temp = value
variable = $(temp) more

當然，您從沒有定義過叫做temp的變數，如您在原始定義變數時，變數值中就引入(include)變數引用，此時可以更爲深刻地體現使用不同模式定義的的重要性。拿下面常見的例子，
CFLAGS = $(includes) -O
...
CFLAGS += -pg # enable profiling
第一行定義了變數CFLAGS，而且變數CFLAGS引用了其它變數，includes。（變數CFLAGS用於C編譯器的規則，參閱隱含規則目錄。）由於定義時使用‘=’，所以變數CFLAGS是遞迴型變數，意味著‘$(includes) -O’在make處理變數CFLAGS定義時是不展開的；*也就是變數includes在生效之前不必定義，它僅需要在任何引用變數CFLAGS之前定義即可*。如果我們試圖不使用‘+=’爲變數CFLAGS附加文字(text)，我們可能按下述模式：
CFLAGS := $(CFLAGS) -pg # enable profiling
這似乎很好，但結果絕不是我們所希望的。使用‘:=’重新定義變數CFLAGS爲簡單展開型變數，意味著make在設定變數CFLAGS之前展開了‘$(CFLAGS) -pg’。*如果變數includes此時沒有定義，我們將得到‘-0 -pg’*，並且以後對變數includes的定義也不會有效。相反，使用‘+=’設定變數CFLAGS我們得到沒有展開的‘$(CFLAGS) 0 -pg’，這樣保留了對變數includes的引用，在後面一個地方如果變數includes得到定義，‘$(CFLAGS)’仍然可以使用它的值。

6.7 撤銷(override)指令

如果一個變數設定時使用了命令參數（參閱變數重載），那麼在makefile檔案中通常的對該變數賦值不會生效。此時對該變數進行設定，您需要使用**撤銷(override)指令**，其格式如下：
override variable = value
或
override variable := value
爲該變數追加更多的文字(text)，使用：
override variable += more text
參閱爲變數值附加文字(text)。
撤銷(override)指令不是打算擴大makefile和命令參數衝突，而是希望用它您可以改變和追加哪些設定時使用了命令參數的變數的值。

例如，假設您在執行C編譯器時總是使用‘-g’開關，但您允許用戶像往常一樣使用命令參數指定其它開關，您就可以使用撤銷(override)指令:
override CFLAGS += -g
您也可以在define指令中使用撤銷(override)指令，下面的例子也許就是您想要得：
override define foo
bar
endef
關於define指令的訊息參閱下節。

6.8定義多行變數

設定變數值的另一種方法時使用define指令。這個指令有一個特殊的用法，既可以定義包含多行字符的變數。這使得定義命令的固定次序十分方便（參閱定義固定次序命令）。

在define指令同一行的後面一般是變數名，當然，也可以什麼也沒有。變數的值由下面的幾行給出，值的結束由僅僅包含 endef 的一行標示出。除了上述在語法上的不同之外，define指令像‘=’一樣工作：它建立了一個遞迴型變數（參閱變數的兩個特色）。變數的名字可以包括函數呼叫和變數引用，它們在指令讀入時展開，以便能夠計算出實際的變數名。

```
define two-lines
echo foo
echo $(bar)
endef
```

變數的值在通常的賦值語句中只能在一行中完成，但在define指令中在define指令行以後endef行之前中間所有的行都是變數值的一部分（最後一行除外，因爲標示endef那一行不能認爲是變數值的一部分）。前面的例次功能上等同於：
two-lines = echo foo; echo $(bar)
因爲兩命令之間用分號隔開，其行爲很接近於兩個分離的shell命令。然而，注意使用兩個分離的行，意味著make請求shell兩次，每一行都在獨立的子shell中執行。參閱執行命令。

如果您希望使用define指令的變數定義比使用命令行定義的變數優先，您可以把define指令和撤銷(override)指令一塊使用：

```
override define two-lines
foo
$(bar)
endef
```

參閱撤銷(override)指令。

6.9 環境變數

make使用的變數可以來自make的執行環境。任何make能夠看見的環境變數，在make開始執行時都轉變爲同名同值的make變數。但是，在makefile檔案中對變數的具體賦值，或使用帶有參數的命令，都可以對環境變數進行重載（如果明確使用‘-e’標誌，環境變數的值可以對makefile檔案中的賦值進行重載，參閱選項概要，但是這在實際中不推薦使用。）

這樣，透過在環境中設定變數CFLAGS，您可以實現下絕大多數makefile檔案中的所有C源程式的編譯使用您選擇的開關。因爲您知道沒有makefile將該變數用於其它任務，所以這種使用標準簡潔含義的變數是安全的（但這也是不可靠的，一些makefile檔案可能設定變數CFLAGS，從而使環境中變數CFLAGS的值失效）。當使用遞迴的make時，在外層make環境中定義的變數，可以傳遞給內層的make（參閱遞迴make）。預設模式下，只有環境變數或在命令行中定義的變數才能傳遞給內層make。您可以使用export指令傳遞其它變數，參閱與子make通訊的變數。

環境變數的其它使用模式都不推薦使用。將makefile的執行完全依靠環境變數的設定、超出makefile檔案的控制範圍，這種做法是不明智的，因爲不同的用戶執行同一個makefile檔案有可能得出不同的結果。這和大部分makefile檔案的意圖相違背。

變數SHELL在環境中存在，用來指定用戶對交互的shell的選擇，因此使用變數SHELL也存字類似的問題。這種根據選定值影響make執行的模式是很不受歡迎的。所以，make將忽略環境中變數SHELL的值（在MS-DOS 和 MS-Windows中執行例外，但此時變數SHELL通常不設定值，參閱執行命令）。

6.10 特定目標變數的值

make中變數的值一般是全域性的；既，無論它們在任何地方使用，它們的值是一樣的（當然，您重新設定除外）；自動變數是一個例外（參閱自動變數）。

另一個例外是特定目標變數的值，這個特點允許您可以根據make建造目標的變化改變變數的定義。像自動變數一樣，這些值只能在一個目標的命令腳本的上下文起作用。

可以像這樣設定特定目標變數的值：

target ... : variable-assignment
或這樣：
target ... : override variable-assignment

'target ...' 中可含有多個目標，如此，則設定的特定目標變數的值可在目標清單中的任一個目標中使用。 'variable-assignment' 使用任何賦值模式都是有效的：遞迴（ '=' ）、靜態（ ':=' ）、追加（ '+=' ）或條件（ '?=' ）。所有出現下 'variable-assignment' 中的變數能夠在特定目標target ...的上下文中使用：也就是任何以前為特定目標target ...定義的特定目標變數的值在這些特定目標中都是有效的。注意這種變數值和全域變數值相比是局部的值：這兩種類型的變數不必有相同的類型（遞迴vs.靜態）。
特定目標變數的值和其它makefile變數具有相同的優先權。一般在命令行中定義的變數（和強製使用 '-e' 情況下的環境變數）的值佔據優先的地位，而使用撤銷(override)指令定義的特定目標變數的值則佔據優先地位。
特定目標變數的值有另外一個特點：當您定義一個特定目標變數時，該變數的值對特定目標target ...的所有先決條件有效，除非這些先決條件用它們自己的特定目標變數的值將該變數重載。例如：

prog : CFLAGS = -g
prog : prog.o foo.o bar.o
將在目標prog的命令腳本中設定變數CFLAGS的值為 '-g' ，同時在建立`prog.o', `foo.o', 和 `bar.o'的命令腳本中變數CFLAGS的值也是 '-g' ，以及prog.o', 'foo.o', 和 'bar.o'的先決條件的建立命令腳本中變數CFLAGS的值也是 '-g' 。

6.11 特定樣式變數的值

除了特定目標變數的值（參閱上小節）外，GNU make也支援特定樣式變數的值。使用特定樣式變數的值，可以為匹配指定格式的目標定義變數。在為目標定義特定目標變數後將搜尋按特定樣式定義的變數，在為該目標的父目標定義的特定目標變數前也要搜尋按特定樣式定義的變數。

設定特定樣式變數格式如下：
pattern ... : variable-assignment
或這樣：
pattern ... : override variable-assignment
這裡的 **'pattern' 是%-格式。像特定目標變數的值一樣，** 'pattern ...' 中可含有多個格式，如此，則設定的特定樣式變數的值可在匹配清單中的任一個格式中的目標中使用。 'variable-assignment' 使用任何賦值模式都是有效的，在命令行中定義的變數的值佔據優先的地位，而使用撤銷(override)指令定義的特定樣式變數的值則佔據優先地位。例如：
%.o : CFLAGS = -O
搜尋所有匹配樣式%.o的目標，並將它的變數CFLAGS的值設定為 '-0' 。

7 makefile檔案的條件語句

一個條件語句可以導致根據變數的值執行或忽略makefile檔案中一部分腳本。**條件語句可以將一個變數與其它變數的值相比較，或將一個變數與一字元串常量相比較。條件語句用於控制make實際看見的makefile檔案部分，不能用於在執行時控制shell命令。**

7.1條件語句的例子

下述的條件語句的例子告訴make如果變數CC的值是‘gcc’時使用一個資料庫，如不是則使用其它資料庫。它透過控制選擇兩命令行之一作爲該規則的命令來工作。‘CC=gcc’作爲make改變的參數的結果不僅用於決定使用哪一個編譯器，而且決定連接哪一個資料庫。

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

該條件語句使用三個指令：ifeq、else和endif。

Ifeq指令是條件語句的開始，並指明條件。它包含兩個參數，它們被逗號分開，並被擴在圓括號內。執行時首先對兩個參數變數替換，然後進行比較。在makefile中跟在ifeq後面的行是符合條件時執行的命令；否則，它們將被忽略。

如果前面的條件失敗，else指令將導致跟在其後面的命令執行。在上述例子中，意味著當第一個選項不執行時，和第二個選項連在一起的命令將執行。在條件語句中，else指令是可選擇使用的。

Endif指令結束條件語句。任何條件語句必須以endif指令結束，後跟makefile檔案中的正常內容。

上例表明條件語句工作在原文水準：條件語句的行根據條件要么被處理成makefile檔案的一部分或要么被忽略。這是makefile檔案重大的語法單位（例如規則）可以跨越條件語句的開始或結束的原因。

當變數CC的值是gcc，上例的效果爲：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(libs_for_gcc)
```

當變數CC的值不是gcc而是其它值的時候，上例的效果爲：

```
foo: $(objects)
        $(CC) -o foo $(objects) $(normal_libs)
```

相同的結果也能使用另一種方法獲得：先將變數的賦値條件化，然後再使用變數：

```
libs_for_gcc = -lgnu
normal_libs =

ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
        $(CC) -o foo $(objects) $(libs)
```

7.2條件語句的語法

對於沒有else指令的條件語句的語法爲：

conditional-directive
text-if-true
endif

‘text-if-true’可以是任何文字(text)行，在條件爲‘真’時它被認爲是makefile檔案的一部分；如果條件爲‘假’，將被忽略。完整的條件語句的語法爲：

conditional-directive

text-if-true
else
text-if-false
endif
如果條件爲‘真’，使用‘text-if-true’；如果條件爲‘假’，使用‘text-if-false’。‘text-if-false’可以是任意多行的文字(text)。

關於‘conditional-directive’的語法對於簡單條件語句和複雜條件語句完全一樣。有四種不同的指令用於測試不同的條件。下面是指令表：
ifeq (arg1, arg2)
ifeq 'arg1' 'arg2'
ifeq "arg1" "arg2"
ifeq "arg1" 'arg2'
ifeq 'arg1' "arg2"

展開參數arg1、arg2中的所有變數引用，並且比較它們。如果它們完全一致，則使用‘text-if-true’，否則使用‘text-if-false’（如果存在的話）。您經常要測試一個變數是否有非空值，當經過複雜的變數和函數展開得到一個值，對於您認爲是空值，實際上有可能由於包含空格而被認爲不是空值，由此可能造成混亂。對於此，您可以使用strip函數從而避免空格作爲非空值的干擾。例如：

```
ifeq ($(strip $(foo)),)
text-if-empty
endif
```

即使$(foo)中含有空格，也使用‘text-if-empty’。
ifneq (arg1, arg2)
ifneq 'arg1' 'arg2'
ifneq "arg1" "arg2"
ifneq "arg1" 'arg2'
ifneq 'arg1' "arg2"
展開參數arg1、arg2中的所有變數引用，並且比較它們。如果它們不同，則使用‘text-if-true’，否則使用‘text-if-false’（如果存在的話）。

ifdef variable-name
如果變數‘variable-name’是非空值，‘text-if-true’有效，否則，‘text-if-false’有效（如果存在的話）。變數從沒有被定義過則變數是空值。**注意ifdef僅僅測試變數是否有值。它不能展開到看變數是否有非空值。因而，使用ifdef測試所有定義過的變數都返回‘真’，但那些像‘foo=’情況除外。測試空值請使用ifeq($(foo),)。例如：**

```
bar =
foo = $(bar)
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

設定‘frobozz'的值爲‘yes', 而：:

```
foo =
ifdef foo
frobozz = yes
else
frobozz = no
endif
```

設定‘frobozz' 爲‘no'。

ifndef variable-name
如果變數‘variable-name’是空值，‘text-if-true’有效，否則，‘text-if-false’有效（如果存在的話）。

在指令行前面允許有多餘的空格，它們在處理時被忽略，但是不允許有Tab（如果一行以Tab開始，那麼該行將被認爲是規則的命令行）。除此之外，空格和Tab可以插入到行的任何地方，當然指令名和參數中間除外。以‘#’開始的註釋可以在行的結尾。

在條件語句中另兩個有影響的指令是else和endif。這兩個指令以一個單字的形式出現，沒有任何參數。在指令行前面允許有多餘的空格，空格和Tab可以插入到行的中間，以‘#’開始的註釋可以在行的結尾。
條件語句影響make使用的makefile檔案。如果條件爲‘真’，make讀入‘text-if-true’包含的行；如果條件爲‘假’，make讀入‘text-if-false’包含的行（如果存在的話）；makefile檔案的語法單位，例如規則，可以跨越條件語句的開始或結束。

當讀入makefile檔案時，Make計算條件的值。因而您不能在測試條件時使用自動變數，因爲他們是命令執行時才被定義（參閱自動變數）。
爲了避免不可忍受的混亂，在一個makefile檔案中開始一個條件語句，而在另外一個makefile檔案中結束這種情況是不允許的。然而如果您試圖引入引入(include)的makefile檔案不中斷條件語句，您可以在條件語句中編寫include指令。

7.3測試標誌的條件語句

您可以使用變數MAKEFLAGS和findstring函數編寫一個條件語句，用它來測試例如‘-t’等的make命令標誌（參閱字元串替換和分析的函數）。這適用於僅使用touch標誌不能完全更改檔案的時間戳的場合。
findstring函數檢查一個字元串是否爲另一個字元串的子字元串。如果您要測試‘-t’標誌，使用‘-t’作爲第一個字元串，將變數MAKEFLAGS的值作爲另一個字元串。例如下面的例子是安排使用‘ranlib t’完成一個資料庫檔案的更新：

```
archive.a: ...
ifneq (,$(findstring t,$(MAKEFLAGS)))
        +touch archive.a
        +ranlib -t archive.a
else
        ranlib archive.a
endif
```

前綴‘+’表示這些命令行是遞迴行，即使是用‘-t’標誌它們一樣要執行。參閱遞迴make。

8 文字(text)轉換函數

函數允許您在makefile檔案中處理文字(text)、計算檔案、操作使用命令等。在函數呼叫時您必須指定函數名以及函數操作使用的參數。函數處理的結果將返回到makefile檔案中的呼叫點，其模式和變數替換一樣。

8.1函數呼叫語法

函數呼叫和變數引用類似，它的格式如下：

$(function arguments)

或這樣：

${function arguments}

這裡‘function’是函數名，是make內建函數清單中的一個。當然您也可以使用建立函數call建立的您自己的函數。‘arguments’是該函數的參數。參數和函數名之間是用空格或Tab隔開，如果有多個參數，它們之間用逗號隔開。這些空格和逗號不是參數值的一部分。包圍函數呼叫的定界符，無論圓括號或大括號，可以在參數中成對出現，在一個函數呼叫中只能有一種定界符。如果在參數中包含變數引用或其它的函數呼叫，最好使用同一種定界符，如寫為‘$(subst a,b,$(x))', 而不是 `$(subst a,b,${x})'。這是因為這種模式不但比較清楚，而且也有在一個函數呼叫中只能有一種定界符的規定。

為每一個參數寫的文字(text)經過變數替換或函數呼叫處理，最終得到參數的值，這些值是函數執行必須依靠的文字(text)。另外，變數替換是按照變數在參數中出現的次序進行處理的。

逗號和不成對出現的圓括號、大括號不能作為文字(text)出現下參數中，前導空格也不能出現下第一個參數中。這些字符不能被變數替換處理為參數的值。如果需要使用這些字符，首先定義變數comma和space，它們的值是單獨的逗號和空格字符，然後在需要的地方因用它們，如下例：

```
comma:= ,
empty:=
space:= $(empty) $(empty)
foo:= a b c
bar:= $(subst $(space),$(comma),$(foo))
# bar is now `a,b,c'.
```

這裡函數subst的功能是將變數foo中的空格用逗號替換，然後返回結果。

8.2字元串替換和分析函數

這裡有一些用於操作字元串的函數：

$(subst from,to,text)

在文字(text)‘text’中使用‘to’替換每一處‘from’。例如：

$(subst ee,EE,feet on the street)

結果為‘fEEt on the street’。

$(patsubst pattern,replacement,text)

尋找‘text’中符合格式‘pattern’的字，用‘replacement’替換它們。這裡‘pattern’中包含萬用字元‘%’，它和一個字中任意個數的字符相匹配。如果‘replacement’中也含有萬用字元‘%’，則這個‘%’被和‘pattern’中萬用字元‘%’匹配的文字(text)代替。在函數patsubst中的‘%’可以用反斜線(\)(‘\’)引用。引用字符‘%’的反斜線(\)可以被更多反斜線(\)引用。引用字符‘%’和其它反斜線(\)的反斜線(\)在比較檔案名或有一個徑(stem)代替它之前從格式中移出。使用反斜線(\)引用字符‘%’不會帶來其它麻煩。例如，格式‘the\%weird\\%pattern\\'是‘the%weird\' 加上萬用字元‘%'然後和字元串‘pattern\\'連接。最後的兩個反斜線(\)由於不能影響任何統配符‘%’所以保持不變。在字之間的空格間被壓縮為單個空格，前導以及結尾空格被丟棄。例如：

$(patsubst %.c,%.o,x.c.c bar.c)

的結果為：‘x.c.o bar.o'。替換引用是實現函數patsubst功能一個簡單方法：

$(var:pattern=replacement)

等同於 ：

$(patsubst pattern,replacement,$(var))

另一個通常使用的函數patsubst的簡單方法是：替換檔案名的後置。
$(var:suffix=replacement)
等同於：
$(patsubst %suffix,%replacement,$(var))

例如您可能有一個OBJ檔案的清單：
objects = foo.o bar.o baz.o
要得到這些檔案的源檔案，您可以簡單的寫爲：
$(objects:.o=.c)
代替規範的格式：
$(patsubst %.o,%.c,$(objects))

$(strip string)
去掉前導和結尾空格，並將中間的多個空格壓縮爲單個空格。這樣，‘$(strip a b c)'結果爲‘a b c’。函數strip和條件語句連用非常有用。當使用ifeq或ifneq把一些值和空字元串‘’比較時，您通常要將一些僅由空格組成的字元串認爲是空字元串（參閱makefile中的條件語句）。如此下面的例子在實現預期結果時可能失敗：

```
.PHONY: all
ifneq   "$(needs_made)" ""
all: $(needs_made)
else
all:;@echo 'Nothing to make!'
endif
```

在條件指令ifneq中用函數呼叫‘$(strip $(needs_made))'代替變數引用‘$(needs_made)'將不再出現問題。

$(findstring find,in)
在字元串‘in’中搜尋‘find’，如果找到，則返回值是‘find’，否則返回值爲空。您可以在一個條件中使用該函數測試給定的字元串中是否含有特定的子字元串。這樣，下面兩個例子：
$(findstring a,a b c)
$(findstring a,b c)
將分別產生值‘a’和‘’。對於函數findstring的特定用法參閱測試標誌的條件語句。

$(filter pattern...,*text*)
返回在‘text’中由空格隔開且匹配樣式‘pattern...’的字，對於不符合格式‘pattern...’的字移出。格式用‘%’寫出，和前面論述過的函數patsubst的格式相同。函數filter可以用來變數分離類型不同的字元串。例如：
```
sources := foo.c bar.c baz.s ugh.h
foo: $(sources)
        cc $(filter %.c %.s,$(sources)) -o foo
```
表明‘foo' 依靠‘foo.c',‘bar.c',‘baz.s' 和‘ugh.h'；但僅有‘foo.c',‘bar.c' 和‘baz.s' 指明用命令編譯。

$(filter-out pattern...,*text*)
返回在‘text’中由空格隔開且不匹配樣式‘pattern...’的字，對於符合樣式‘pattern...’的字移出。只是函數filter的反函數。例如：
```
objects=main1.o foo.o main2.o bar.o
mains=main1.o main2.o
```
下面產生不引入(include)在變數‘mains’中的OBJ檔案的檔案清單：
$(filter-out $(mains),$(objects))

$(sort list)
將‘list’中的字按字母順序排序，並取掉重複的字。輸出是由單個空格隔開的字的清單。
$(sort foo bar lose)
返回值是‘bar foo lose’。順便提及，由於函數sort可以取掉重複的字，您就是不關心排序也可以使用它的這個特點。
這裡有一個實際使用函數subst和patsubst的例子。假設一個makefile檔案使用變數VPATH指定make搜尋先決條件檔案的一系列路徑(stem)（參閱VPATH:先決條件搜尋路徑(stem)）。這個例子表明怎樣告訴C編譯器在相同路徑(stem)清單中搜尋頭檔案。

變數VPATH的值是一列用冒號隔開的路徑(stem)名，如‘src:../headers'。首先，函數subst將冒號變爲空格：
$(subst :, ,$(VPATH))
這產生值‘src ../headers'。然後，函數patsubst爲每一個路徑(stem)名加入‘-I’標誌，這樣這些路徑(stem)可以加到變數CFLAGS中，就可以自動傳遞給C編譯器：
override CFLAGS += $(patsubst %,-I%,$(subst :, ,$(VPATH)))
結果是在以前給定的變數CFLAGS的值後附加文字(text)‘-Isrc -I../headers’。撤銷(override)指令的作用是即使以前使用命令參數指定變數CFLAGS的值，新値也能起作用。參閱撤銷(override)指令。

8.3檔案名函數

其中幾個內建的展開函數和拆分檔案名以及列舉檔案名相關聯。下面列舉的函數都能執行對檔案名的特定轉換。函數的參數是一系列的檔案名，檔案名之間用空格隔開（前導和結尾空格被忽略）。清單中的每一個檔案名都採用相同的模式轉換，而且結果用單個空格串聯在一起。

$(dir names...)
抽取‘names’中每一個檔案名的路徑(stem)部分，檔案名的路徑(stem)部分包括從檔案名的開始到最後一個斜線(/)（含斜線(/)）之前的一切字符。如果檔案名中沒有斜線(/)，路徑(stem)部分是‘./’。如：
$(dir src/foo.c hacks)
產生的結果爲 ‘src/ ./’。

$(notdir names...)
抽取‘names’中每一個檔案名中除路徑(stem)部分外一切字符（真正的檔案名）。如果檔案名中沒有斜線(/)，則該檔案名保持不變，否則，將路徑(stem)部分移走。一個檔案名如果僅引入(include)路徑(stem)部分（以斜線(/)結束的檔案名）將變爲空字元串。這是非常不幸的，因爲這意味著在結果中如果有這種檔案名存在，兩檔案名之間的空格將不是由相同多的空格隔開。但現下我們並不能看到其它任何有效的代替品。例如：
$(notdir src/foo.c hacks)
產生的結果爲‘foo.c hacks’。

$(suffix names...)
抽取‘names’中每一個檔案名的後置。如果檔案名中（或含有斜線(/)，且在最後一個斜線(/)後）含有句點，則後置是最後那個句點以後的所有字符，否則，後置是空字元串。如果結果爲空意味著‘names’沒有帶後置檔案名，如果檔案中含有多個檔案名，則結果列出的後置數很可能比原檔案名數目少。例如：
$(suffix src/foo.c src-1.0/bar.c hacks)
產生的結果是‘.c .c’。

$(basename names...)
抽取‘names’中每一個檔案名中除後置外一切字符。如果檔案名中（或含有斜線(/)，且在最後一個斜線(/)後）含有句點，則基本名字是從開始到最後一個句點（不引入(include)）間的所有字符。如果沒有句點，基本名字是整個檔案名。例如：
$(basename src/foo.c src-1.0/bar hacks)
產生的結果爲‘src/foo src-1.0/bar hacks’。

$(addsuffix suffix,names...)
參數‘names’作爲一系列的檔案名，檔案名之間用空格隔開；suffix作爲一個單位。將Suffix（後置）的值附加在每一個獨立檔案名的後面，完成後將檔案名串聯起來，它們之間用單個空格隔開。例如：
$(addsuffix .c,foo bar)
結果爲‘foo.c bar.c’。

$(addprefix prefix,names...)
參數‘names’作爲一系列的檔案名，檔案名之間用空格隔開；prefix作爲一個單位。將preffix（前綴）的值附加在每一個獨立檔案名的前面，完成後將檔案名串聯起來，它們之間用單個空格隔開。例如：
$(addprefix src/,foo bar)
結果爲‘src/foo src/bar’。

$(join list1,list2)

將兩個參數串聯起來：兩個參數的第一個字串聯起來形成結果的第一個字，兩個參數的第二個字串聯起來形成結果的第二個字，以此類推。如果一個參數比另一個參數的字多，則多餘的字原封不動的拷貝到結果上。例如，'$(join a b,.c .o)'產生'a.c b.o'。字之間多餘的空格不再保留，它們由單個空格代替。該函數可將函數dir、notdir的結果合併，產生原始給定的檔案清單。

$(word n,text)

返回'text'中的第n個字。N的合法值從1開始。如果n比'text'中的字的數目大，則返回空值。例如：

$(word 2, foo bar baz)

返回'bar'。

$(wordlist s,e,text)

返回'text'中的從第s個字開始到第e個字結束的一列字。S、e的合法值從1開始。如果s比'text'中的字的數目大，則返回空值；如果e比'text'中的字的數目大，則返回從第s個字開始到'text'結束的所有字；如果s比e大，不返回任何值。例如：

$(wordlist 2, 3, foo bar baz)

返回`bar baz'。

$(words text)

返回'text'中字的數目。這樣'text'中的最後一個字是'$(word $(words text),text)'。

$(firstword names...)

參數'names'作為一系列的檔案名，檔案名之間用空格隔開；返回第一個檔案名，其餘的忽略。例如：

$(firstword foo bar)

產生結果'foo'。 雖然 $(firstword text) 和 $(word 1,text)的作用相同，但第一個函數因為簡單而保留下來。

$(wildcard pattern)

參數'pattern'是一個檔案名樣式，典型的引入(include)萬用字元（和shel中的檔案名一樣）。函數wildcard的結果是一列和樣式匹配的且檔案存在的檔案名，檔案名之間用一個空格隔開，參閱在檔案名中使用萬用字元。

8.4函數foreach

函數foreach和其它函數非常不同，它導致一個文字(text)塊重複使用，而且每次使用該文字(text)塊進行不同的替換；它和shell sh中的命令for及C-shell csh中的命令foreach類似。

函數foreach語法如下：

$(foreach var,list,text)

前兩個參數，'var'和'list'，將首先展開，注意最後一個參數'text'此時不展開；接著，對每一個'list'展開產生的字，將用來為'var'展開後命名的變數賦值；然後'text'引用該變數展開；因此它每次展開都不相同。

結果是由空格隔開的'text' 在'list'中多次展開的字組成的新的'list'。'text'多次展開的字串聯起來，字與字之間由空格隔開，如此就產生了函數foreach的返回值。

這是一個簡單的例子，將變數'files'的值設定為 'dirs'中的所有目錄下的所有檔案的清單：

```
dirs := a b c d
files := $(foreach dir,$(dirs),$(wildcard $(dir)/*))
```

這裡'text'是'$(wildcard $(dir)/*)'。第一個為變數dir發現的值是'a'，所以產生函數foreach結果的第一個字為'$(wildcard a/*)'；第二個重複的值是'b'，所以產生函數foreach結果的第二個字為'$(wildcard b/*)'；第三個重複的值是'c'，所以產生函數foreach結果的第三個字為'$(wildcard c/*)'；等等。該例子和下例有共同的結果：

```
files := $(wildcard a/* b/* c/* d/*)
```

如果'text'比較複雜，您可以使用附加變數為它命名，這樣可以提升程式的可讀性：

```
find_files = $(wildcard $(dir)/*)
dirs := a b c d
files := $(foreach dir,$(dirs),$(find_files))
```

這裡我們使用變數find_file。我們定義變數find_file時，使用了‘=’，因此該變數爲遞迴型變數，這樣變數find_file所引入(include)的函數呼叫將在函數foreach控制下在展開；對於簡單展開型變數將不是這樣，在變數find_file定義時就呼叫函數wildcard。
函數foreach對變數‘var’沒有長久的影響，它的值和變數特色在函數foreach呼叫結束後將和前面一樣，其它從‘list’得到的值僅在函數foreach執行時起作用，它們是暫時的。變數‘var’在函數foreach執行期間是簡單展開型變數，如果在執行函數foreach之前變數‘var’沒有定義，則函數foreach呼叫後也沒有定義。參閱變數的兩個特色。
當使用複雜變數表達式產生變數名時應特別小心，因爲許多奇怪的字符作爲變數名是有效的，但很可能不是您所需要的，例如：
files := $(foreach Esta escrito en espanol!,b c ch,$(find_files))
如果變數find_file展開引用名爲‘Esta escrito en espanol!’變數，上例是有效的，但它極易帶來錯誤。

8.5函數if

函數if對在函數上下文中展開條件提供了支援（相對於GNU make makefile檔案中的條件語句，例如ifeq指令，參閱條件語句的語法）。
一個函數if的呼叫，可以引入(include)兩個或三個參數：
$(if condition,then-part[,else-part])
第一個參數‘condition’，首先把前導、結尾空格去掉，然後展開。如果展開爲非空字元串，則條件‘condition’爲‘真’；如果展開爲空字元串，則條件‘condition’爲‘假’。
如果條件‘condition’爲‘真’,那麼計算第二個參數‘then-part’的值，並將該值作爲整個函數if的值。
如果條件‘condition’爲‘假’,第三個參數如果存在，則計算第三個參數‘else-part’的值，並將該值作爲整個函數if的值；如果第三個參數不存在，函數if將什麼也不計算，返回空值。
注意僅能計算‘then-part’和‘else-part’二者之一，不能同時計算。這樣有可能產生副作用（例如函數shell的呼叫）。

8.6函數call

函數call是唯一的建立新的帶有參數函數的函數。您可以寫一個複雜的表達是作爲一個變數的值，然後使用函數call用不同的參數呼叫它。
函數call的語法爲：
$(call variable,param,param,...)
當make展開該函數時，它將每一個參數‘param’賦值給臨時變數$(1)、$(2)等；變數$(0)的值是變數‘variable’。對於參數‘param’的數量無沒有最大數目限制，也沒有最小數目限制，但是如果使用函數call而沒有任何參數，其意義不大。變數‘variable’在這些臨時變數的上下文中被展開爲一個make變數，這樣，在變數‘variable’中對變數‘$(1)’的引用決定了呼叫函數call時對第一個參數‘param’的使用。

注意變數‘variable’是一個變數的名稱，不是對該變數的引用，所以，您不能採用‘$’和圓括號的格式書寫該變數，當然，如果您需要使用非常量的檔案名，您可以在檔案名中使用變數引用。
如果變數名是內建函數名，則該內建函數將被呼叫（即使使用該名稱的make變數已經存在）。函數call在給臨時變數賦值以前首先展開參數，這意味著，變數‘variable’對內建函數的呼叫採用特殊的規則進行展開，像函數foreach或if，它們的展開結果和您預期的結果可能不同。下面的一些例子能夠更清楚的表達這一點。
該例子時使用巨集將參數的順序翻轉：

```
reverse = $(2) $(1)

foo = $(call reverse,a,b)
```

這裡變數foo的值是‘b a’。
下面是一個很有意思的例子：它定義了一個巨集，使用該巨集可以搜尋變數PATH引入(include)的所有目錄中的第一個指定類型的程式：

```
pathsearch = $(firstword $(wildcard $(addsufix /$(1),$(subst :, ,$(PATH)))))

LS := $(call pathsearch,ls)
```

現下變數LS的值是‘/bin/ls’或其它的類似的值。
在函數call中可以使用巢狀。每一次遞迴都可以爲它自己的局部變數‘$(1)’等賦值，從而代替上一層函數call賦的值。例如：這實現了映像函數功能。
map = $(foreach a,$(2),$(call $(1),$(a)))
現下您可以映像（map）僅有一個參數的函數，如函數origin，一步得到多個值：

o = $(call map,origin,o map MAKE)
最後變數o引入(include)諸如‘file file default’這樣的值。
警告：在函數call的參數中使用空格一定要十分小心。因為在其它函數中，第二個或接下來的參數中的空格是不刪除的，這有可能導致非常奇怪的結果。當您使用函數call時，去掉參數中任何多餘的空格才是最安全的方法。

8.7函數origin

函數origin不像一般函數，它不對任何變數的值操作；它僅僅告訴您一些關於一個變數的訊息；它特別的告訴您變數的來源。
函數origin的語法：
$(origin variable)
注意變數‘variable’是一個查詢變數的名稱，不是對該變數的引用所以，您不能採用‘$’和圓括號的格式書寫該變數，當然，如果您需要使用非常量的檔案名，您可以在檔案名中使用變數引用。
函數origin的結果是一個字元串，該字元串變數是怎樣定義的：
‘undefined'
如果變數‘variable’從沒有定義。
‘default'
變數‘variable’是預設定義，通常和命令CC等一起使用，參閱隱含規則使用的變數。注意如果您對一個預設變數重新進行了定義，函數origin將返回後面的定義。
‘environment'
變數‘variable’作為環境變數定義，選項‘-e’沒有打開（參閱選項概要）。
‘environment override'
變數‘variable’作為環境變數定義，選項‘-e’已打開（參閱選項概要）。
‘file'
變數‘variable’在makefile中定義。
‘command line'
變數‘variable’在命令行中定義。
‘override'
變數‘variable’在makefile中用撤銷(override)指令定義（參閱撤銷(override)指令）。
‘automatic'
變數‘variable’是自動變數，定義它是為了執行每個規則中的命令（參閱自動變數）。

這種訊息的基本用途（其它用途是滿足您的好奇心）是使您要了解變數值的依據。例如，假設您有一個名為‘foo’的makefile檔案，它引入(include)了另一個名為‘bar’的makefile檔案，如果在環境變數中已經定義變數‘bletch’，您希望執行命令‘make f bar’在makefile檔案‘bar’中重新定義變數‘bletch’。但是makefile檔案‘foo’在包括makefile檔案‘bar’之前已經定義了變數‘bletch’，而且您也不想使用撤銷(override)指令定義，那麼您可以在makefile檔案‘foo’中使用撤銷(override)指令，因為撤銷(override)指令將會重載任何命令行中的定義，所以其定義的優先權超越以後在makefile檔案‘bar’中的定義。因此makefile檔案‘bar’可以引入(include)：

```
ifdef bletch
ifeq "$(origin bletch)" "environment"
bletch = barf, gag, etc.
endif
endif
```

如果變數‘bletch’在環境中定義，這裡將重新定義它。
即使在使用選項‘-e’的情況下，您也要對來自環境的變數‘bletch’重載定義，則您可以使用如下內容：

```
ifneq "$(findstring environment,$(origin bletch))" ""
bletch = barf, gag, etc.
endif
```

如果‘$(origin bletch)’返回‘environment’或‘environment override’，這裡將對變數‘bletch’重新定義。參閱字元串替換和分析函數。

8.8 函數shell

除了函數wildcard之外，函數shell和其它函數不同，它是make與外部環境的通訊工具。函數shell和在大多數shell中後引號（’）執行的功能一樣：它用於命令的展開。這意味著它起著呼叫shell命令和返回命令輸出結果的參數的作用。Make僅

僅處理返回結果，再返回結果替換呼叫點之前，make將每一個換行符或者一對返回/換行符處理爲單個空格；如果返回結果最後是換行符（和返回符），make將把它們去掉。由函數shell呼叫的命令，一旦函數呼叫展開，就立即執行。在大多數情況下，當makefile檔案讀入時函數shell呼叫的命令就已執行。例外情況是在規則命令行中該函數的呼叫，因爲這種情況下只有在命令執行時函數才能展開，其它呼叫函數shell的情況和此類似。

這裡有一些使用函數shell的例子：
contents := $(shell cat foo)
將含有檔案foo的目錄設定爲變數contents的值，是用空格（而不是換行符）分離每一行。
files := $(shell echo *.c)
將‘*.c’的展開設定爲變數files的值。除非make使用非常怪異的shell，否則這條語句和‘wildcard *.c’的結果相同。

8.9 控制make的函數

這些函數控制make的執行模式。通常情況下，它們用來向用戶提供makefile檔案的訊息或在偵測到一些類型的環境錯誤時中斷make執行。

$(error text...)
通常‘text’是致命的錯誤訊息。注意錯誤是在該函數計算時產生的，因此如果您將該函數放在命令的腳本中或遞迴型變數賦值的右邊，它直到過期也不能計算。‘text’將在錯誤產生之前展開，例如：
ifdef ERROR1
$(error error is $(ERROR1))
endif
如果變數ERROR01已經定義，在將makefile檔案讀入時產生致命的錯誤。或，
ERR = $(error found an error!)

.PHONY: err
err: ; $(ERR)
如果err目標被呼叫，在make執行時產生致命錯誤。

$(warning text...)
該函數和函數error工作的模式類似，但此時make不退出，即雖然‘text’展開並顯示結果訊息，但make仍然繼續執行。展開該函數的結果是空字元串。

9 執行make

講述編譯程式的makefile檔案，可以由多種模式實現。最簡單的模式是編譯所有過期的檔案，對於通常所寫的makefile檔案，如果不使用任何參數執行make，那麼將這樣執行。

但是您也許僅僅更新一部分檔案；您也許需要使用不同的編譯器或不同的編譯選項；您也許僅僅希望找出舊式的檔案而不更新它們。這些只有透過在執行make時給出參數才能實現。退出make狀態有三種情況：

0
表示make成功完成退出。
2
退出狀態爲2表示make執行中遇到錯誤，它將列印訊息描述錯誤。
1
退出狀態爲1表示您執行make時使用了‘-q’標誌，並且make決定一些檔案沒有更新。參閱代替執行命令。

9.1 指定makefile檔案的參數

指定makefile檔案名的方法是使用‘-f’或‘--file’選項（‘--makefile’也能工作）。例如，‘-f altmake’說明名爲‘altmake’的檔案作爲makefile檔案。

如果您連續使用‘-f’標誌幾次，而且每一個‘-f’後面都帶有參數，則所有指定的檔案將連在一起作爲makefile檔案。

如果您不使用‘-f’或‘--file’選項，預設的是按次序尋找‘GNUmakefile', ‘makefile', 和 ‘Makefile'，使用這三個中第一個能夠找到的存在檔案或能夠建立的檔案，參閱編寫makefile檔案。

9.2指定最終目標的參數

最終目標（gaol）是make最終努力更新的目標。其它更新的目標是因爲它們作爲最終目標的先決條件，或先決條件的先決條件，等等以此類推。

預設情況下，makefile檔案中的第一個目標是最終目標（不計算那些以句點開始的目標）。因此，makefile檔案的第一個編譯目標是對整個程式或程式組描述。如果第一個規則同時擁有幾個目標，只有該規則的第一個目標是預設的最終目標。

您可以使用make的參數指定最終目標。方法是使用目標的名字作爲參數。如果您指定幾個最終目標，make按您命名時的順序一個接一個的處理它們。

任何在makefile檔案中出現的目標都能作爲最終目標（除了以‘-’開始或含有‘=’的目標，它們一種解析爲開關，另一種是變數定義）。即使在makefile檔案中沒有出現的目標，按照隱含規則可以說明怎樣生成，也能指定爲最終目標。

Make將在命令行中使用特殊變數MAKECMGOALS設定您指定的最終目標。如果沒有在命令行指定最終目標，該變數的值爲空值。注意該變數值能在特殊場合下使用。

一個合適的例子是在清除規則中避免刪除包括‘.d’的檔案（參閱自動產生先決條件），因這樣make不會一建立它們，就立即又刪除它們：

```
sources = foo.c bar.c

ifneq ($(MAKECMDGOALS),clean)
include $(sources:.c=.d)
endif
```

指定最終目標的一個用途是僅僅編譯程式的一部分或程式組中的幾個程式。如是這樣，您可以將您希望變異的檔案指定爲最終目標。例如，在一個路徑(stem)下引入(include)幾個程式，一個makefile檔案以下面的格式開始：

```
.PHONY: all
all: size nm ld ar as
```

如果您僅對程式size編譯，則您可以使用‘make size’命令，這樣就只有您指定的程式才重新編譯。

指定最終目標的另一個用途是編譯產生哪些沒有正常生成的檔案。例如，又一個檔案需要調試，或一個版本的程式需要編譯進行測試，然而該檔案不是makefile檔案規則中預設最終目標的先決條件，此時，可以使用最終目標參數指定它們。

指定最終目標的另一個用途是執行和一個假想(phony)目標（參閱假想(phony)目標）或空目標（使用空目標記錄事件）相聯繫的命令。許多makefile檔案引入(include)一個假想(phony)目標‘clean’刪除除了原檔案以外的所有檔案。正常情況下，只有您具體指明使用‘make clean’命令，make才能執行上述任務。下面列出典型的假想(phony)目標和空目標的名稱。對GNU make套裝軟件使用的所有標準目標名參閱用戶標準目標：

‘all'

建立makefile檔案的所有頂層目標。
`clean'
刪除所有make正常建立的檔案。
`mostlyclean'
像假象目標‘clean’，但避免刪除人們正常情況下不重新建造的一少部分檔案。例如，用於GCC的目標‘mostlyclean’不刪除‘libgcc.a’，因爲重建它的情況十分稀少，而且建立它又需要很多時間。
`distclean'
`realclean'
`clobber'
這些目標可能定義爲比目標‘clean’刪除更多的檔案。例如，刪除配置檔案或爲編譯正常建立的準備檔案，甚至makefile檔案自身不能建立的檔案。
‘install’
向命令搜尋目錄下拷貝可執行檔案；向可執行檔案尋找目錄下拷貝可執行檔案使用的輔助檔案。
‘print’
列印發生變化的檔案清單。
‘tar’
建立源檔案的壓縮‘tar’檔案。
‘shar’
爲源檔案建立一個shell的資料庫檔案。
‘dist’
爲源檔案建立一個發布檔案。這可能是‘tar’檔案，‘shar’檔案，或多個上述的壓縮版本檔案。
‘TAGS’
更新該程式的‘tags’標籤。
`check'
`test'
對該makefile檔案建立的程式執行自我測試。

9.3 代替執行命令

makefile檔案告訴make怎樣識別一個目標是否需要更新以及怎樣更新每一個目標。但是更新目標並不是您一直需要的，一些特定的選項可以用來指定make的其它活動：
`-n'
`--just-print'
`--dry-run'
`--recon'
‘No-op’。make的這項活動是列印用於建立目標所使用的命令，但並不執行它們。
`-t'
`--touch'
‘touch’。這項活動是做更新標誌，實際卻不更改它們。換句話說，make假裝編譯了目標，但實際對它們沒有一點兒改變。
`-q'
`--question'
‘question’。這項活動是暗中察看目標是否已經更新；但是任何情況下也不執行命令。換句話說，即不編譯也不輸出。
`-W file'
`--what-if=file'
`--assume-new=file'
`--new-file=file'
‘What if’。每一個‘-W’標誌後跟一個檔案名。所有檔案名的更改時間被make記錄爲當前時間，但實際上更改時間保持不變。如果您要更新檔案，您可以使用‘-W’標誌和‘-n’標誌連用看看將發生什麼。
使用標誌‘-n’，make列印那些正常執行的命令，但卻不執行它們。
使用標誌‘-t’，make忽略規則中的命令，對那些需要更新的目標使用‘touch’命令。如果不使用‘-s’或.SILENT，‘touch’命令同樣列印。爲了提升執行效率，make並不實際呼叫程式touch，而是使touch直接執行。
使用標誌‘-q’，make不列印輸出也不執行命令，如果所有目標都已經更新到最新，make的退出狀態是0；如果一部分需要更新，退出狀態是1；如果make遇到錯誤，退出狀態是2，因此您可以根據沒有更新的目標尋找錯誤。

在執行make時對以上三個標誌如果同時兩個或三個將產生錯誤。標誌‘-n’、‘-t’和‘-s’對那些以字符‘+’開始的命令行和引入(include)字元串‘$(MAKE)' 或‘${MAKE}'命令行不起作用。注意僅有這些以字符‘+’開始的命令行和引入(include)字元串‘$(MAKE)' 或‘${MAKE}'命令行執行時不注意這些選項。參閱變數MAKE的工作模式。

- ‘-W’標誌有一下兩個特點：
- 如果同時使用標志‘-n’或‘-q’，如果您更改一部分文件，看看make將會做什么。
- 沒有使用標志‘-n’或‘-q’，如果make執行時采用標志‘-W’，則make假裝所有文件已經更新，但實際上不更改任何文件。

注意選項‘-p’和‘-v’允許您得到更多的make訊息或正在使用的makefile檔案的訊息（參閱選項概要）。

9.4避免重新編譯檔案

有時您可能改變了一個源檔案，但您並不希望編譯所有依靠它的檔案。例如，假設您在一個許多檔案都依靠的頭檔案種添加了一個巨集或一個聲明，按照保守的規則，make認爲任何對於該頭檔案的改變，需要編譯所有依靠它的檔案，但是您知道那是不必要的，並且您沒有等待它們完全編譯的時間。
如果您提前了解改變頭檔案以前的問題，您可以使用‘-t’選項。該標誌告訴make不執行規則中的命令，但卻將所有目標的時間戳改到最新。您可按下述步驟實現上述計畫：
1、用make命令重新編譯那些需要編譯的源檔案；
2、更改頭檔案；
3、使用‘make t’命令改變所有目標檔案的時間戳，這樣下次執行make時就不會因爲頭檔案的改變而編譯任何一個檔案。
如果在重新編譯那些需要編譯的源檔案前已經改變了頭檔案，則按上述步驟做已顯得太晚了；作爲補救措施，您可以使用‘-o file’標誌，它能將指定的檔案的時間戳假裝改爲以前的時間戳（參閱選項概要）。這意味著該檔案沒有更改，因此您可按下述步驟進行：
1、使用‘make -o file’命令重新編譯那些不是因爲改變頭檔案而需要更新的檔案。如果涉及幾個頭檔案，您可以對每個頭檔案都使用‘-o’標誌進行指定。
2、使用‘make t’命令改變所有目標檔案的時間戳。

9.5變數重載

使用‘=’定義的變數：‘v=x’將變數v的値設爲x。如果您用該方法定義了一個變數，在makefile檔案後面任何對該變數的普通賦値都將被make忽略，要使它們生效應在命令行將它們重載。
最爲常見的方法是使用傳遞附加標誌給編譯器的靈活性。例如，在一個makefile檔案中，變數CFLAGS已經引入(include)了執行C編譯器的每一個命令，因此，如果僅僅鍵入命令make時，檔案‘foo.c’將按下面的模式編譯：
cc -c $(CFLAGS) foo.c
這樣您在makefile檔案中對變數CFALAGS設定的任何影響編譯器執行的選項都能生效，但是每次執行make時您都可以將該變數重載，例如：如果您說‘make CFLAGS='-g -O'’，任何C編譯器都將使用‘cc -c -g -O’編譯程式。這還說明了在重載變數時，怎樣使用shell命令中的引用包括空格和其它特殊字符在內的變數的値。
變數CFALAGS僅僅是您可以使用這種模式重載的許多標準變數中的一個，這些標準變數的完整清單見隱含規則使用的變數。
您也可以編寫makefile察看您自己的附加變數，從而使用戶可透過更改這些變數控制make執行時的其它面貌。
當您使用命令參數重載變數時，您可以定義遞迴展開型變數或簡單展開型變數。上例中定義的是遞迴展開型變數，如果定義簡單展開型變數，請使用‘:=’代替‘=’。注意除非您在變數値中使用變數引用或函數呼叫，這兩種變數沒有任何差異。
利用這種模式也可以改變您在makfile檔案中重載的變數。在makfile檔案中重載的變數是使用撤銷(override)指令，是和‘override variable = value’相似的命令行。詳細內容參閱撤銷(override)指令。

9.6 測試編譯程式

正常情況下，在執行shell命令時一旦有錯誤發生，make立即退出返回非零狀態；不會爲任何目標繼續執行命令。錯誤表明make不能正確的建立最終目標，並且make一發現錯誤就立即報告。
當您編譯您修改過的程式時，這不是您所要的結果。您希望make能夠經可能的試著編譯每一個程式，並儘可能的顯示每一個錯誤。
這種情況下，您可以使用**‘-k’或‘--keep-going’選項。這種選項告訴make遇到錯誤返回非零狀態之前，繼續尋找該目標的先決條件，如果有必要則重新建立它們**。例如，在編譯一個目標檔案時發現錯誤，即使make已經知道連接它們已是不可能的, ‘make k’也將繼續編譯其它目標檔案。除在shell命令失敗後繼續執行外，即使發在make不知道如何建立

的目標和先決條件檔案以後，‘make k’也將儘可能的繼續執行。在沒有‘-k’選項時，這些錯誤將是致命的（參閱選項概要）。
通常情況下，make的行爲是基於假設您的目標是使最終目標更新；一旦它發現這是不可能的它就立即報告錯誤。選項‘-k’說真正的目標是儘可能測試改變對程式的影響，發現存在的問題，以便在下次執行之前您可以糾正它們。這是Emacs M-x compile命令預設傳遞‘-k’選項的原因。

9.7 選項概要

下面是所有make能理解的選項清單：

`-b'
`-m'
和其它版本make兼容時，這些選項被忽略。

`-C dir'
`--directory=dir'
在將makefile讀入之前，把路徑(stem)切換到‘dir’下。如果指定多個‘-C’選項，每一個都是相對於前一個的解釋：‘-C/-C etc’等同於‘-C/etc’。該選項典型用在遞迴make過程中，參閱遞迴make。

‘-d’
在正常處理後列印調試訊息。調試訊息說明哪些檔案用於更新，哪個檔案作爲比較時間戳的標準以及比較的結果，哪些檔案實際上需要更新，需要考慮、使用哪些隱含規則等等----一切和make決定最終干什麼有關的事情。‘-d’選項等同於‘--debug=a’選項（參見下面內容）。
`--debug[=options]'
在正常處理後列印調試訊息。可以選擇各種級別和類型的輸出。如果沒有參數，列印‘基本’級別的調試訊息。以下是可能的參數，僅僅考慮第一個字母，各個值之間使用逗號或空格隔開：

a (all)
顯示所有調試訊息，該選項等同於‘-d’選項。

b (basic)
基本調試訊息列印每一個已經舊式的目標，以及它們重建是否成功。

v (verbose)
比‘基本’級高一個的等級的調試訊息。包括makefile檔案的語法分析結果，沒有必要更新的先決條件等。該選項同時引入(include)基本調試訊息。

i (implicit)
列印隱含規則搜尋目標的訊息。該選項同時引入(include)基本調試訊息。

j (jobs)
列印各種子命令呼叫的詳細訊息。

m (makefile)
以上選項不引入(include)重新建立makefile檔案的訊息。該選項引入(include)了這方面的訊息。注意，選項‘all’也不引入(include)這方面訊息。該選項同時引入(include)基本調試訊息。

`-e'
`--environment-overrides'
設定從環境中繼承來的變數的優先權高於makefile檔案中的變數的優先權。參閱環境變數。

`-f file'
`--file=file'
`--makefile=file'

將名爲‘file’的檔案設定爲makefile檔案。參閱編寫makefile檔案。

`-h'
`--help'
向您提醒make 能夠理解的選項，然後退出。

`-i'
`--ignore-errors'
忽略重建檔案執行命令時產生的所有錯誤。

`-I dir'
`--include-dir=dir'
指定搜尋引入(include)makefile檔案的路徑(stem)‘dir’。參閱引入(include)其它makefile檔案。如果同時使用幾個‘-I’選項用於指定路徑(stem)，則按照指定的次序搜尋這些路徑(stem)。

`-j [jobs]'
`--jobs[=jobs]'
指定同時執行的命令數目。如果沒有參數make將同時執行儘可能多的任務；如果有多個‘-j’選項，則僅最後一個選項有效。詳細內容參閱並行執行。注意在MS-DOS下，該選項被忽略。

`-k'
`--keep-going'
在出現錯誤後，儘可能的繼續執行。當一個目標建立失敗，則所有依靠它的目標檔案將不能重建，而這些目標的其它先決條件則可繼續處理。參閱測試編譯程式。

`-l [load]'
`--load-average[=load]'
`--max-load[=load]'
指定如果有其它任務正在執行，並且平均負載已接近或超過‘load’（一個浮點數），則此時不啓動新任務。如果沒有參數則取消以前關於負載的限制。參閱並行執行。

`-n'
`--just-print'
`--dry-run'
`--recon'
列印要執行的命令，但卻不執行它們。參閱代替執行命令。

`-o file'
`--old-file=file'
`--assume-old=file'
即使檔案file比它的先決條件‘舊’，也不重建該檔案。不要因爲檔案file的改變而重建任何其它檔案。該選項本質上是假裝將該檔案的時間戳改爲舊的時間戳，以至於依靠它的規則被忽略。參閱避免重新編譯檔案。

`-p'
`--print-data-base'
列印資料庫（規則和變數的值），這些數據來自讀入makefile檔案的結果；然後像通常那樣執行或按照別的指定選項執行。如果同時給出‘-v’開關，則列印版本訊息（參閱下面內容）。使用‘make qp’則列印資料庫後不試圖重建任何檔案。使用‘make p f/dev/null’則列印預定義的規則和變數的資料庫。資料庫輸出中引入(include)檔案名，以及命令和變數定義的行號訊息。它是在複雜環境中很好的調試工具。

`-q'
`--question'
‘問題模式’。不列印輸出也不執行命令，如果所有目標都已經更新到最新，make的退出狀態是0；如果一部分需要更

新，退出狀態是1；如果make遇到錯誤，退出狀態是2，參閱代替執行命令。

`-r'
`--no-builtin-rules'
排除使用內建的隱含規則（參閱使用隱含規則）。您仍然可以定義您自己的樣式規則（參閱定義和重新定義樣式規則）。選項‘-r’同時也清除了預設的後置清單和後置規則(suffix rule)（參閱舊式的後置規則(suffix rule)）。但是您可以使用.SUFFIXES規則定義您自己的後置。注意，使用選項‘-r’僅僅影響規則；預設變數仍然有效（參閱隱含規則使用的變數）；參閱下述的選項‘-R’。
`-R'
`--no-builtin-variables'
排除使用內建的規則變數（參閱隱含規則使用的變數）。當然，您仍然可以定義自己的變數。選項‘-R’自動使選項‘-r’生效；因為它去掉了隱含規則所使用的變數的定義，所以隱含規則也就失去了存在的意義。

`-s'
`--silent'
`--quiet'
沈默選項。不回顯那些執行的命令。參閱命令回顯。

`-S'
`--no-keep-going'
`--stop'
使選項‘-k’失效。除非在遞迴make時，透過變數MAKEFLAGS從上層make繼承選項‘-k’，或您在環境中設定了選項‘-k’，否則沒有必要使用該選項。

`-t'
`--touch'
標誌檔案已經更新到最新，但實際沒有更新它們。這是假裝那些命令已經執行，用於愚弄將來的make呼叫。參閱代替執行命令。

`-v'
`--version'
列印make程式的版本訊息，作者清單和沒有擔保的注意訊息，然後退出。

`-w'
`--print-directory'
列印執行makefile檔案時涉及的所有工作目錄。這對於跟蹤make遞迴時複雜巢狀產生的錯誤非常有用。參閱遞迴make。實際上，您很少需要指定該選項，因為make已經替您完成了指定。參閱‘--print-directory’選項。

`--no-print-directory'
在指定選項‘-w’的情況下，禁止列印工作路徑(stem)。這個選項在選項‘-w’自動打開而且您不想看多餘訊息時比較有用。參閱‘--print-directory’選項。

`-W file'
`--what-if=file'
`--new-file=file'
`--assume-new=file'
假裝目標檔案已經更新。在使用標誌‘n’時，它將向您表明更改該檔案會發生什麼。如果沒有標誌‘n’它和在執行make之前對給定的檔案使用touch命令的結果幾乎一樣，但使用該選項make只是在的想像中更改該檔案的時間戳。參閱代替執行命令。

`--warn-undefined-variables'
當make看到引用沒有定義的變數時，發布一條警告訊息。如果您按照複雜模式使用變數，當您調試您的makefile檔案時，該選項非常有用。

10 使用隱含規則
重新建立目標檔案的一些標準方法是經常使用的。例如，一個傳統的建立OBJ檔案的方法是使用C編譯器，如cc，編譯C語言源程式。

隱含規則能夠告訴make怎樣使用傳統的技術完成任務，這樣，當您使用它們時，您就不必詳細指定它們。例如，有一條編譯C語言源程式的隱含規則，檔案名決定執行哪些隱含規則；另如，編譯C語言程式一般是使用‘.c’檔案，產生‘.o’檔案。因此， make據此和檔案名的後置就可以決定使用編譯C語言源程式的隱含規則。一系列的隱含規則可按順序應用；例如，make可以從一個‘.y’檔案，借助‘.c’檔案，重建一個‘.o’檔案，參閱隱含規則鏈。內建隱含規則的命令需要使用變數，透過改變這些變數的值，您就可以改變隱含規則的工作模式。例如，變數CFLAGS控制隱含規則用於編譯C程式傳遞給C編譯器的標誌，參閱隱含規則使用的變數。透過編寫樣式規則，您可以建立您自己的隱含規則。參閱定義和重新定義樣式規則。

後置規則(suffix rule)是對定義隱含規則最有限制性。樣式規則一般比較通用和清楚，但是後置規則(suffix rule)卻要保持兼容性。參閱舊式的後置規則(suffix rule)。

10.1 使用隱含規則
允許make對一個目標檔案尋找傳統的更新方法，您所有做的是避免指定任何命令。可以編寫沒有命令行的規則或根本不編寫任何規則。這樣，make將根據存在的源檔案的類型或要生成的檔案類型決定使用何種隱含規則。
例如，假設makefile檔案是下面的格式：

```
foo : foo.o bar.o
        cc -o foo foo.o bar.o $(CFLAGS) $(LDFLAGS)
```

因爲您提及了檔案‘foo.o’，但是您沒有給出它的規則，make將自動尋找一條隱含規則，該規則能夠告訴make怎樣更新該檔案。無論檔案‘foo.o’存在與否，make都會這樣執行。
如果能夠找到一條隱含規則，則它就能夠對命令和一個或多個先決條件（源檔案）提供支援。如果您要指定附加的先決條件，例如頭檔案，但隱含規則不能支援，您需要爲目標‘foo.o’寫一條不帶命令行的規則。
每一條隱含規則都有目標樣式和先決條件格式；也許多條隱含規則有相同的目標樣式。例如，有數不清的規則產生‘.o’檔案：使用C編譯器編譯‘.C’檔案；使用Pascal編譯器編譯‘.p’檔案；等等。實際應用的規則是那些先決條件存在或可以建立的規則。所以，如果您有一個‘.C’檔案，make將執行C編譯器；如果您有一個‘.p’檔案，make將執行Pascal編譯器；等等。
當然，您編寫一個makefile檔案時，您知道您要make使用哪一條隱含規則，以及您知道make將選擇哪一條規則，因爲您知道那個先決條件檔案是假設存在的。預定義的隱含規則清單的詳細內容參閱隱含規則目錄。
首先，我們說一條隱含規則可以應用，該規則的先決條件必須‘存在或可以建立’。一個檔案‘可以建立’是說該檔案在makefile中作爲目標或先決條件被提及，或者該檔案可以經過一條隱含規則的遞迴後能夠建立。如果一條隱含規則的先決條件是另一條隱含規則的結果，我們說產生了‘鏈’。參閱隱含規則鏈。
總體上說，make爲每一個目標搜尋隱含規則，爲沒有命令行的**雙冒號規則(::)**搜尋隱含規則。僅作爲先決條件被提及的檔案，將被認爲是一個目標，如果該目標的規則沒有指定任何內容， make將爲它搜尋隱含規則。對於詳細的搜尋過程參閱隱含規則的搜尋算法。
注意，任何具體的先決條件都不影響對隱含規則的搜尋。例如，認爲這是一條具體的規則：

```
foo.o: foo.p
```

檔案foo.p不是首要條件，這意味著make按照隱含規則可以從一個Pascal源程式（‘.p’檔案）建立OBJ檔案，也就是說一個‘.o’檔案可根據‘.p’檔案進行更新。但檔案foo.p並不是絕對必要的；例如，如果檔案foo.c也存在，按照隱含規則則是從檔案foo.c重建foo.o，這是因爲C編譯規則在預定義的隱含規則清單中比Pascal規則靠前，參閱隱含規則目錄。
如果您不希望使用隱含規則建立一個沒有命令行的目標，您可以透過添加分號爲該目標指定空命令。參閱使用空命令。

10.2隱含規則目錄
這裡列舉了預定義的隱含規則的目錄，這些隱含規則是經常應用的，當然如果您在makefile檔案中重載或刪除後，這些隱含規則將會失去作用，詳細內容參閱刪除隱含規則。選項‘-r’或‘--no-builtin-rules’將刪除所有預定義的隱含規則。
並不是所有的隱含規則都是預定義的，在make中很多預定義的隱含規則是後置規則(suffix rule)的展開，因此，那些預定義的隱含規則和後置規則(suffix rule)的清單相關（特殊目標.SUFFIXES的先決條件清單）。預設的後置清單爲：.out, .a, .ln, .o, .c, .cc, .C, .p, .f, .F, .r, .y, .l, .s, .S, .mod, .sym, .def, .h, .info, .dvi, .tex, .texinfo, .texi, .txinfo, .w, .ch，.web, .sh, .elc, .el。所有下面描述的隱含規則，如果它們的先決條件中有一個出現下這個後置清單中，則是後置規則(suffix

rule)。如果您更改這個後置清單，則只有那些由一個或兩個出現下您指定的清單中的後置命名的預定義後置規則(suffix rule)起作用；那些後置沒有出現下清單中的規則被禁止。對於詳細的關於後置規則(suffix rule)的描述參閱舊式的後置規則(suffix rule)。

Compiling C programs（編譯C程式）

‘n.o' 自動由‘n.c' 使用命令 ‘$(CC) -c $(CPPFLAGS) $(CFLAGS)'生成 。

Compiling C++ programs （編譯C++程式）

‘n.o'自動由‘n.cc' 或‘n.C'使用命令‘$(CXX) -c $(CPPFLAGS) $(CXXFLAGS)’ 生成。 我們鼓勵您對C++源檔案使用後置‘.cc' 代替後置‘.C’ 。

Compiling Pascal programs （編譯Pascal程式）

‘n.o'自動由‘n.p'使用命令‘$(PC) -c $(PFLAGS)'生成。

Compiling Fortran and Ratfor programs （編譯Fortran 和 Ratfor程式）

‘n.o'自動由‘n.r', ‘n.F'或‘n.f' 執行Fortran編譯器生成。使用的精確命令如下：

`.f'

`$(FC) -c $(FFLAGS)'.

`.F'

`$(FC) -c $(FFLAGS) $(CPPFLAGS)'.

`.r'

`$(FC) -c $(FFLAGS) $(RFLAGS)'.

Preprocessing Fortran and Ratfor programs （預處理Fortran 和 Ratfor程式）

‘n.f' 自動從 ‘n.r'或‘n.F'得到。該規則僅僅是與處理器把一個Ratfor 程式或能夠預處理的 Fortran 程式轉變爲標準的 Fortran 程式。使用的精確命令如下：

`.F'

`$(FC) -F $(CPPFLAGS) $(FFLAGS)'.

`.r'

`$(FC) -F $(FFLAGS) $(RFLAGS)'.

Compiling Modula-2 programs（編譯Modula-2程式）

‘n.sym'自動由‘n.def'使用命令‘$(M2C) $(M2FLAGS) $(DEFFLAGS)'生成。 ‘n.o' 從 ‘n.mod'生成；命令爲:‘$(M2C) $(M2FLAGS) $(MODFLAGS)'。

Assembling and preprocessing assembler programs （彙編以及預處理組合程式）

‘n.o'自 ‘n.S'執行C編譯器，cpp，生成。命令爲:‘$(CPP) $(CPPFLAGS)'。

Linking a single object file （連接一個簡單的OBJ檔案）

‘n' 自動由‘n.o' 執行C編譯器中的連接程式 linker (通常稱爲 ld)生成。命令爲: ‘$(CC) $(LDFLAGS) n.o $(LOADLIBES) $(LDLIBS)'。該規則對僅有一個源程式的簡單程式或對同時含有多個OBJ檔案（可能來自於不同的源檔案）的程式都能正常工作。如果同時含有多個OBJ檔案，則其中必有一個OBJ檔案的名字和可執行檔案名匹配。例如：

x: y.o z.o

當‘x.c', ‘y.c' 和‘z.c' 都存在時則執行：

cc -c x.c -o x.o

cc -c y.c -o y.o

cc -c z.c -o z.o

cc x.o y.o z.o -o x

rm -f x.o

rm -f y.o

rm -f z.o

對於更複雜的情況，例如沒有一個OBJ檔案的名字和可執行檔案名匹配，您必須爲連接寫一條具體的命令。每一種能自動生成‘.o’的檔案，可以在沒有‘-c’選項的情況下使用編譯器(‘$(CC)',‘$(FC)'或‘$(PC)'； C編譯器‘$(CC)'也適用於組合程式)自動連接。當然也可以使用OBJ檔案作爲中間檔案，但編譯、連接一步完成速度將快很多。

Yacc for C programs （由Yacc生成C程式）

‘n.c'自動由‘n.y'使用命令‘$(YACC) $(YFLAGS)'執行 Yacc生成。

Lex for C programs （由Lex生成C程式）

‘n.c'自動由‘n.l' 執行 Lex生成。命令爲：‘$(LEX) $(LFLAGS)'。

Lex for Ratfor programs （由Lex生成Rator程式）

‘n.r'自動由‘n.l' 執行 Lex生成。命令爲：‘$(LEX) $(LFLAGS)'。 對於所有的Lex檔案，無論它們產生C代碼或Ratfor 代碼，都使用相同的後置‘.l’進行轉換，在特定場合下，使用make自動確定您使用哪種語言是不可能的。如果make使用‘.l’ 檔案重建一個OBJ檔案，它必須猜想使用哪種編譯器。它很可能猜想使用的是 C 編譯器, 因爲C 編譯器更加普遍。

如果您使用 Ratfor語言, 請確保在makefile檔案中提及了‘n.r'，使make知道您的選擇。否則，如果您專用Ratfor語言，不使用任何C 檔案, 請在隱含規則後置清單中將‘.c’剔除：
.SUFFIXES:
.SUFFIXES: .o .r .f .l ...
Making Lint Libraries from C, Yacc, or Lex programs（由C, Yacc, 或 Lex程式建立Lint庫）
‘n.ln' 可以從‘n.c' 執行lint產生。命令爲：‘ $(LINT) $(LINTFLAGS) $(CPPFLAGS) i’。用於C程式的命令和用於‘n.y'或‘n.l'程式相同。
TeX and Web（TeX 和 Web）
‘n.dvi'可以從‘n.tex' 使用命令‘$(TEX)'得到。‘n.tex'可以從‘n.web'使用命令‘$(WEAVE)'得到；或者從‘n.w' (和‘n.ch'，如果‘n.ch'存在或可以建造) 使用命令‘$(CWEAVE)'。‘n.p' 可以從‘n.web'使用命令‘$(TANGLE)'產生。‘n.c' 可以從‘n.w' (和‘n.ch'，如果‘n.ch'存在或可以建造) 使用命令‘$(CTANGLE)'得到。
Texinfo and Info（Texinfo和Info）
‘n.dvi' 可以從‘n.texinfo',‘n.texi', 或‘n.txinfo', 使用命令‘$(TEXI2DVI) $(TEXI2DVI_FLAGS)' 得到。‘n.info' 可以從‘n.texinfo',‘n.texi', 或‘n.txinfo', 使用命令‘$(MAKEINFO) $(MAKEINFO_FLAGS)'建立。
RCS
檔案‘n'必要時可以從名爲‘n,v'或‘RCS/n,v'的RCS檔案中提取。具體命令是：‘$(CO) $(COFLAGS)'。檔案‘n'如果已經存在，即使RCS檔案比它新，它不能從RCS檔案中提取。用於RCS的規則是最終的規則，參閱萬用規則，所以RCS不能夠從任何源檔案產生，它們必須存在。
SCCS
檔案‘n'必要時可以從名爲‘s.n'或‘SCCS/s.n'的SCCS檔案中提取。具體命令是：‘$(GET) $(GFLAGS)'。用於SCCS的規則是最終的規則，參閱萬用規則，所以SCCS不能夠從任何源檔案產生，它們必須存在。SCCS的優點是，檔案‘n' 可以從檔案‘n.sh'拷貝並生成可執行檔案(任何人都可以)。這用於shell的腳本，該腳本在SCCS內部檢查。因爲RCS 允許保持檔案的可執行性，所以您沒有必要將該特點用於RCS檔案。我們推薦您避免使用SCCS，RCS不但使用廣泛，而且是免費的軟體。選擇自由軟體代替相當的（或低劣的）收費軟體，是您對自由軟體的支援。
通常情況下，您要僅僅改變上表中的變數，需要參閱下面的文檔。

隱含規則的命令實際使用諸如COMPILE.c, LINK.p, 和 PREPROCESS.S等等變數，它們的值引入(include)以上列出的命令。Make按照慣例進行處理，如，編譯‘.x’源檔案的規則使用變數‘COMPILE.x’；從‘.x’源檔案生成可執行檔案使用變數‘LINK.x’；預處理‘.x’源檔案使用變數‘PREPROCESS.x’。

任何產生OBJ檔案的規則都使用變數‘OUTPUT_OPTION’；make依據編譯時間選項定義該變數的值是‘-o $@’或空值。當源檔案分佈在不同的目錄中，您應該使用‘-O’選項保證輸出到正確的檔案中；使用變數VPATH時同樣（參閱爲先決條件搜尋目錄）。一些系統的編譯器不接受針對OBJ檔案的‘-o’開關；如果您在這樣的系統上執行，並使用了變數VPATH，一些檔案的編譯輸出可能會放到錯誤的地方。解決辦法是將變數OUTPUT_OPTION值設爲：‘; mv $*.o $@’。

10.3隱含規則使用的變數

內建隱含規則的命令對預定義變數的使用是開放的；您可以在makefile檔案中改變變數的值，也可以使用make的執行參數或在環境中改變，如此，在不對這些規則本身重新定義的情況下，就可以改變這些規則的工作模式。您還可以使用選項‘-R’或‘--no-builtin-variables’刪除所有隱含規則使用的變數。

例如，編譯C程式的命令實際是‘$(CC) -c $(CFLAGS) $(CPPFLAGS)’，變數預設的值是‘cc’或空值，該命令實際是‘cc c’。如重新定義變數‘CC’的值爲‘ncc’，則所有隱含規則將使用‘ncc’作爲編譯C語言源程式的編譯器。透過重新定義變數‘CFLAGS’的值爲‘-g’，則您可將‘-g’選項傳遞給每個編譯器。所有的隱含規則編譯C程式時都使用‘$CC’獲得編譯器的名稱，並且都在傳遞給編譯器的參數中都引入(include)‘$(CFLAGS)’。

隱含規則使用的變數可分爲兩類：一類是程式名變數（像cc），另一類是引入(include)程式執行參數的變數（像CFLAGS）。(‘程式名’可能也引入(include)一些命令參數，但是它必須以一個實際可以執行的程式名開始。) 如果一個變數值中引入(include)多個參數，它們之間用空格隔開。

這裡是內建規則中程式名變數清單：

AR 檔案管理程式；預設爲：‘ar'.

AS	彙編編譯程式；預設爲：‘as'.
CC	C語言編譯程式；預設爲：‘cc'.
CXX	C++編譯程式；預設爲：‘g++'.
CO	從RCS檔案中解壓縮抽取檔案程式；預設爲：‘co'.
CPP	帶有標準輸出的C語言預處理程式；預設爲：‘$(CC) -E'.
FC	Fortran 以及 Ratfor 語言的編譯和預處理程式；預設爲：‘f77'.
GET	從SCCS檔案中解壓縮抽取檔案程式；預設爲：‘get'.
LEX	將 Lex 語言轉變爲 C 或 Ratfor程式的程式；預設爲：‘lex'.
PC	Pascal 程式編譯程式；預設爲：‘pc'.
YACC	將 Yacc語言轉變爲 C程式的程式；預設爲：‘yacc'.
YACCR	將 Yacc語言轉變爲 Ratfor程式的程式；預設爲：‘yacc -r'.
MAKEINFO	將Texinfo 源檔案轉換爲訊息檔案的程式；預設爲：‘makeinfo'.
TEX	從TeX源產生TeX DVI檔案的程式；預設爲：‘tex'.
TEXI2DVI	從Texinfo源產生TeX DVI 檔案的程式；預設爲：‘texi2dvi'.
WEAVE	將Web翻譯成TeX的程式；預設爲：‘weave'.
CWEAVE	將CWeb翻譯成TeX的程式；預設爲：‘cweave'.
TANGLE	將Web翻譯成 Pascal的程式；預設爲：‘tangle'.
CTANGLE	將Web翻譯成C的程式；預設爲：‘ctangle'.
RM	刪除檔案的命令；預設爲：‘rm -f'.

這裡是值爲上述程式附加參數的變數清單。在沒有注明的情況下，所有變數的值爲空值。

ARFLAGS	用於檔案管理程式的標誌，預設爲：‘rv'.
ASFLAGS	用於彙編編譯器的額外標誌 (當具體呼叫‘.s'或‘.S'檔案時)。
CFLAGS	用於C編譯器的額外標誌。
CXXFLAGS	用於C++編譯器的額外標誌。
COFLAGS	用於RCS co程式的額外標誌。
CPPFLAGS	用於C預處理以及使用它的程式的額外標誌 (C和 Fortran 編譯器)。
FFLAGS	用於Fortran編譯器的額外標誌。
GFLAGS	用於SCCS get程式的額外標誌。
LDFLAGS	用於呼叫linker（‘ld’）的編譯器的額外標誌。
LFLAGS	用於Lex的額外標誌。
PFLAGS	用於Pascal編譯器的額外標誌。
RFLAGS	用於處理Ratfor程式的Fortran編譯器的額外標誌。
YFLAGS	用於Yacc的額外標誌。Yacc。

10.4 隱含規則鏈

有時生成一個檔案需要使用多個隱含規則組成的序列。例如，從檔案‘n.y’生成檔案‘n.o’，首先執行隱含規則Yacc，其次執行規則cc。這樣的隱含規則序列稱爲隱含規則鏈。

如果檔案‘n.c’存在或在makefile檔案中提及，則不需要任何特定搜尋：make首先發現透過C編譯器編譯‘n.c’可生成該OBJ檔案，隨後，考慮生成‘n.c’時，則使用執行Yacc的規則。這樣可最終更新‘n.c’和‘n.o’。

即使在檔案‘n.c’不存在或在makefile檔案中沒有提及的情況下，make也能想像出在檔案‘n.y’和‘n.o’缺少連接！這種情況下，‘n.c’稱爲中間檔案。一旦make決定使用中間檔案，它將把中間檔案輸入資料庫，好像中間檔案在makefile檔案中提及一樣；按照隱含規則的描述建立中間檔案。

中間檔案和其它檔案一樣使用自己的規則重建，但是中間檔案和其它檔案相比有兩種不同的處理模式。

第一個不同的處理模式是如果中間檔案不存在make的行爲不同：平常的檔案b如果不存在，make認爲一個目標依靠檔案b，它總是建立檔案b，然後根據檔案b更新目標；但是檔案b若是中間檔案，make很可能不管它而進行別的工作，即不建立檔案b，也不更新最終目標。只有在檔案b的先決條件比最終目標‘新’時或有其它原因時，才更新最終目標。

第二個不同點是make在更新目標建立檔案b後，如果檔案b不再需要，make將把它刪除。所以一個中間檔案在make執行之前和make執行之後都不存在。Make向您報告刪除時列印一條‘rm　f’命令，表明有檔案被刪除。

通常情況下，任何在makefile檔案中提及的目標和先決條件都不是中間檔案。但是，您可以特別指定一些檔案爲中間檔案，其方法爲：將要指定爲中間檔案的檔案作爲特殊目標 .INTERMEDIATE的先決條件。這種方法即使對採用別的方法

具體提及的檔案也能生效。

您透過將檔案標誌爲secondary檔案可以阻止自動刪除中間檔案。這時，您將您需要保留的中間檔案指定爲特殊目標.SECONDARY的先決條件即可。對於secondary檔案，make不會因爲它不存在而去建立它，也不會自動刪除它。secondary檔案必須也是中間檔案。
您可以列舉一個隱含規則的目標樣式（例如%.o）作爲特殊目標 .PRECIOUS的先決條件，這樣您就可以保留那些由隱含規則建立的檔案名匹配該格式的中間檔案。參閱中斷和關閉make。
一個隱含規則鏈至少包含兩個隱含規則。例如，從‘RCS/foo.y,v’建立檔案‘foo’需要執行RCS、Yacc和cc，檔案foo.y和foo.c是中間檔案，在執行結束後將被刪掉。
沒有一條隱含規則可以在隱含規則鏈中出現兩次以上（含兩次）。這意味著，make不會簡單的認爲從檔案‘foo.o.o’建立檔案foo不是執行linker兩次。這還可以強製make在搜尋一個隱含規則鏈時阻止無限循環。
一些特殊的隱含規則可優化隱含規則鏈控制的特定情況。例如，從檔案foo.c建立檔案foo可以被擁有編譯和連接的規則鏈控制，它使用foo.o作爲中間檔案。但是對於這種情況存在一條特別的規則，使用簡單的命令cc可以同時編譯和連接。因爲優化規則在規則表中的前面，所以優化規則和一步一步的規則鏈相比，優先使用優化規則。

10.5定義與重新定義樣式規則

您可以透過編寫樣式規則定義隱含規則。該規則看起來和普通規則類似，不同之處在於樣式規則的目標中包含字符‘%’（只有一個）。目標是匹配檔案名的格式；字符‘%’可以匹配任何非空的字元串，而其它字符僅僅和它們自己相匹配。先決條件用‘%’表示它們的名字和目標名關聯。
格式‘%.o : %.c’是說將任何‘stem.c’檔案編譯爲‘stem.o’檔案。
在樣式規則中使用的‘%’展開是在所有變數和函數展開以後進行的，它們是在makefile檔案讀入時完成的。參閱使用變數和轉換文字(text)函數。

10.5.1樣式規則簡介

樣式規則是在目標中引入(include)字符‘%’（只有一個）的規則，其它方面看起來和普通規則相同。目標是可以匹配檔案名的格式，字符‘%’可以匹配任何非空的字元串，而其它字符僅僅和它們自己相匹配。
例如‘%.c’匹配任何以‘.c’結尾的檔案名；‘s.%.c’匹配以‘s.’開始並且以‘.c’結尾的檔案名，該檔案名至少引入(include)5個字符（因爲‘%’至少匹配一個字符）。匹配‘%’的子字元串稱爲徑(stem)。先決條件中使用‘%’表示它們的名字中含有和目標名相同的stem。要使用樣式規則，檔案名必須匹配目標的格式，而且符合先決條件格式的檔案必須存在或可以建立。下面規則：

```
%.o : %.c ; command...
```

表明要建立檔案‘n.o’，使用‘n.c’作爲它的先決條件，而且檔案‘n.c’ 必須存在或可以建立。
在樣式規則中，先決條件有時不含有‘%’。這表明採用該樣式規則建立的所有檔案都是採用相同的先決條件。這種固定先決條件的樣式規則在有些場合十分有用。

樣式規則的先決條件不必都引入(include)字符‘%’，這樣的規則是一個有力的常規萬用字元，它爲任何匹配該目標樣式規則的檔案提供建立方法。參閱定義最新類型的預設規則。
樣式規則可以有多個目標，不像正常的規則，這種規則不能扮演具有相同先決條件和命令的多條不同規則。如果一樣式規則具有多個目標，make知道規則的命令對於所有目標來說都是可靠的，這些命令只有在建立所目標時才執行。當爲匹配一目標搜尋樣式規則時，規則的目標樣式和規則要匹配的目標不同是十分罕見的，所以make僅僅擔心目前對檔案給出命令和先決條件是否有問題。注意該檔案的命令一旦執行，所有目標的時間戳都會更新。

樣式規則在makefile檔案中的次序很重要，因爲這也是考慮它們的次序。對於多個都能使用的規則，使用最先出現的規則。您親自編寫的規則比內建的規則優先。注意先決條件存在或被提及的規則優先於先決條件需要經過隱含規則鏈生成的規則。

10.5.2樣式規則的例子

這裡有一些實際在make中預定義的樣式規則例子，第一個，編譯‘.c’檔案生成‘.o’檔案的規則：

```
%.o : %.c
        $(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

定義了一條編譯‘x.c’檔案生成‘x.o’檔案的規則，命令使用自動變數‘$@’和‘$<’替換任何情況使用該規則的目標檔案和源檔案。參閱自動變數。
第二個內建的例子：

```
% :: RCS/%,v
        $(CO) $(COFLAGS) $<
```

定義了在子目錄‘RCS’中根據相應檔案‘x.v’生成檔案‘x’的規則。因爲目標是‘%’，只要相對應的先決條件檔案存在，該規則可以應用於任何檔案。雙冒號表示該規則是最終規則，它意味著不能是中間檔案。參閱萬用規則。
下面的樣式規則有兩個目標：

```
%.tab.c %.tab.h: %.y
        bison -d $<
```

這告訴make執行命令‘bison -d x.y’將建立‘x.tab.c’和‘x.tab.h’。如果檔案foo依靠檔案‘parse.tab.o’和‘scan.o’，而檔案‘scan.o’又依靠檔案‘parse.tab.h’，當‘parse.y’發生變化，命令‘bison -d parse.y’執行一次。‘parse.tab.o’和‘scan.o’的先決條件也隨之更新。（假設檔案‘parse.tab.o’由檔案‘parse.tab.c’編譯生成，檔案‘scan.o’由檔案‘scan.c’生成，當連接‘parse.tab.o’、‘scan.o’和其它先決條件生成檔案foo時，上述規則能夠很好執行。）

10.5.3自動變數

假設您編寫一個編譯‘.c’檔案生成‘.o’檔案的規則：您怎樣編寫命令‘CC’，使它能夠操作正確的檔案名？您當然不能將檔案名直接寫進命令中，因爲每次使用隱含規則操作的檔案名都不一樣。
您應該使用make的另一個特點，自動變數。這些變數在規則每次執行時都基於目標和先決條件產生新值。例如您可以使用變數‘$@’代替目標檔案名，變數‘$<’代替先決條件檔案名。
下面是自動變數清單：

$@
規則的目標檔案名。如果目標是一個資料庫成員，則變數‘$@’ 資料庫檔案的檔案名。對於有多個目標的樣式規則（參閱樣式規則簡介），變數‘$@’是那個導致規則命令執行的目標檔案名。

$%
當目標是資料庫成員時，該變數是目標成員名，參閱使用make更新資料庫檔案。例如，如果目標是‘foo.a(bar.o)'，則‘$%'的值是‘bar.o'，‘$@'的值是‘foo.a'。如果目標不是資料庫成員，則‘$%'是空值。

$<
第一個先決條件的檔案名。如果目標更新命令來源於隱含規則，該變數的值是隱含規則添加的第一個先決條件。參閱使用隱含規則。

$?
所有比目標‘新’的先決條件名，名字之間用空格隔開。對於爲資料庫成員的先決條件，只能使用已命名的成員。參閱使用make更新資料庫檔案。

$^
所有先決條件的名字，名字之間用空格隔開。對於爲資料庫成員的先決條件，只能使用已命名的成員。參閱使用make更新資料庫檔案。對同一個目標來說，一個檔案只能作爲一個先決條件，不管該檔案的檔案名在先決條件清單中出現多少次。所以，如果在先決條件清單中，同一個檔案名出現多次，變數‘$^’的值仍然僅引入(include)該檔案名一次。

$+
該變數像‘$^'，但是，超過一次列出的先決條件將按照它們在makefile檔案中出現的次序複製。這主要的用途是對於在按照特定順序重複庫檔案名很有意義的地方使用連接命令。

$*
和隱含規則匹配的徑(stem)，參閱樣式匹配。如果一個目標爲‘dir/a.foo.b'，目標樣式規則爲：‘a.%.b'，則stem爲‘dir/foo'。在構建相關檔案名時stem 十分有用。在靜態樣式規則中，stem是匹配目標樣式中字符‘%’的檔案名中那一部分。在一個沒有stem具體規則中；變數‘$*' 不能以該方法設定。如果目標名以一種推薦的後置結尾（參閱舊式的後置規則(suffix rule)），變數‘$*'設定爲目標去掉該後置後的部分。例如，如果目標名是‘foo.c'，則變數‘$*' 設定爲‘foo',

因爲‘.c' 是一個後置。GNU make 處理這樣奇怪的事情是爲了和其它版本的make兼容。在隱含規則和靜態樣式規則以外，您應該盡量避免使用變數‘$*'。在具體規則中如果目標名不以推薦的後置結尾，則變數‘$*’在該規則中設定爲空值。

當您希望僅僅操作那些改變的先決條件，變數‘$?' 即使在具體的規則中也很有用。例如，假設名爲‘lib’的資料庫檔案包含幾個OBJ檔案的拷貝，則下面的規則僅將發生變化的OBJ檔案拷貝到資料庫檔案：

```
lib: foo.o bar.o lose.o win.o
        ar r lib $?
```

在上面列舉的變數中，有四個值是單個檔案名。三個值是檔案名清單。這七個存放檔案的路徑(stem)名或存放目錄下檔案名的變體。變數的變體名是由變數名追加字母‘D’或‘F’構成。這些變體在GNU make中處於半廢狀態，原因是使用函數T dir和notdir 能夠得到相同的結果。參閱檔案名函數。注意，‘F'變體省略所有在dir函數中總是輸出的結尾斜線(/) 這裡是這些變體的清單：

`$(@D)'
目標檔案名中的路徑(stem)部分，結尾斜線(/) 已經移走。如果變數`$@'的值是`dir/foo.o'，變體 `$(@D)'的值是`dir'。如果變數`$@'的值不包括斜線(/) ，則變體的值是`.'。
`$(@F)'
目標檔案名中的真正檔案名部分。如果變數`$@'的值是`dir/foo.o'，變體 `$(@F)'的值是` foo.o '。`$(@F)' 等同於 `$(notdir $@)'。
`$(*D)'
`$(*F)'
徑(stem)中的路徑(stem)名和檔案名；在這個例子中它們的值分別爲：`dir' 和 `foo' 。
`$(%D)'
`$(%F)'
資料庫成員名中的路徑(stem)名和檔案名；這僅對採用‘archive(member)’形式的資料庫成員目標有意義，並且當成員引入(include)路徑(stem)名時才有用。參閱資料庫成員目標。

`$(<D)'
`$(<F)'
第一個先決條件名中的路徑(stem)名和檔案名。
`$(^D)'
`$(^F)'
所有先決條件名中的路徑(stem)名和檔案名清單。
`$(?D)'
`$(?F)'
所有比目標‘新’的先決條件名中的路徑(stem)名和檔案名清單。
注意，在我們討論自動變數時，我們使用了特殊格式的慣例；我們寫"the value of‘$<'", 而不是"the variable <" ；和我們寫普通變數，例如變數 objects 和 CFLAGS一樣。我們認爲這種慣例在這種情況下看起來更加自然。這並沒有其它意義，變數‘$<'的變數名爲 < 和變數‘$(CFLAGS)' 實際變數名爲CFLAGS一樣。您也可以使用‘$(<)'代替‘$<'。

10.5.4樣式匹配

目標樣式是由前綴、後置和它們之間的萬用字元%組成，它們中的任一個或兩個都可以是空值。樣式匹配一個檔案名只有該檔案名是以前綴開始，後置結束，而且兩者不重疊的條件下，才算匹配。前綴、後置之間的文字 (text)成爲徑(stem）。當樣式‘%.o’匹配檔案名‘test.o’時，徑(stem)是‘test’。樣式規則中的先決條件將徑(stem)替換字符%，從而得出檔案名。對於上例中，如果一個先決條件爲‘%.c’，則可展開爲‘test.c’。

當目標樣式中不包含斜線(/) （實際並不是這樣），則檔案名中的路徑(stem)名首先被去除，然後，將其和格式中的前綴和後置相比較。在比較之後，以斜線(/) 結尾的路徑(stem)名，將會加在根據樣式規則的先決條件規則產生的先決條件前面。只有在尋找隱含規則時路徑(stem)名才被忽略，在應用時路徑(stem)名絕不能忽略。
例如，‘e%t’和檔案名‘src/eat’匹配，徑(stem)是‘src/a’。當先決條件轉化爲檔案名時，stem中的路徑(stem)名將加在前面，徑(stem)的其餘部分替換‘%’。使用徑(stem) ‘src/a’和先決條件樣式規則‘c%r’匹配得到檔案名‘src/car’。

10.5.5萬用規則match-anything rules.

當一個樣式規則的目標僅僅爲‘%’，它可以匹配任何檔案名，我們稱這些規則爲萬用規則。它們非常有用，但是make使用它們的耗時也很多，因爲make必須爲作爲目標和作爲先決條件列出的每一個檔案都考慮這樣的規則。

假設makefile檔案提及了檔案foo.c。爲了建立該目標，make將考慮是透過連接一個OBJ檔案‘foo.c.o’建立，或是透過使用一步的C編譯連接程式從檔案foo.c.c建立，或是編譯連接Pascal程式foo.c.p建立，以及其它的可能性等。

我們知道make考慮的這些可能性是很可笑的，因爲foo.c就是一個C語言源程式，不是一個可執行程式。如果make考慮這些可能性，它將因爲這些檔案諸如foo.c.o和foo.c.p等都不存在最終拒絕它們。但是這些可能性太多，所以導致make的執行速度極慢。
爲了加快速度，我們爲make考慮匹配萬用規則的模式設定了限制。有兩種不同類型的可以應用的限制，在您每次定義一個萬用規則時，您必須爲您定義的規則在這兩種類型中選擇一種。

一種選擇是標誌該萬用規則是最終規則，即在定義時使用雙冒號定義。一個規則爲最終規則時，只有在它的先決條件存在時才能應用，即使先決條件可以由隱含規則建立也不行。換句話說，在最終規則中沒有進一步的鏈。
例如，從RCS和SCCS檔案中抽取原檔案的內建的隱含規則是最終規則，則如果檔案‘foo.c,v' 不存在，make絕不會試圖從一個中間檔案‘foo.c,v.o’或‘RCS/SCCS/s.foo.c,v’在建立它。 RCS 和 SCCS 檔案一般都是最終源檔案，它不能從其它任何檔案重新建立，所以，make可以記錄時間戳，但不尋找重建它們的模式。

如果您不將萬用規則標誌爲最終規則，那麼它就是非最終規則。一個非最終萬用規則不能用於指定特殊類型數據的檔案。如果存在其它規則（非萬用規則）的目標匹配一檔案名，則該檔案名就是指定特殊類型數據的檔案名。
例如，檔案名‘foo.c' 和樣式規則 `%.c : %.y' (該規則執行Yacc)。無論該規則是否實際使用(如果碰巧存在檔案‘foo.y’，該規則將執行)，和目標匹配的事實就能足夠阻止任何非最終萬用規則在檔案foo.c上使用。這樣，make 考慮就不試圖從檔案‘foo.c.o',‘foo.c.c',‘foo.c.p'等建立可執行的‘foo.c'。
內建的特殊僞樣式規則是用來認定一些特定的檔案名，處理這些檔案名的檔案時不能使用非最終萬用規則。這些僞樣式規則沒有先決條件和命令，它們用於其它目的時被忽略。例如，內建的隱含規則：

%.p :
存在可以保證Pascal源程式如‘foo.p' 匹配特定的目標樣式，從而阻止浪費時間尋找‘foo.p.o' 或‘foo.p.c'。
在後置規則(suffix rule)中，爲後置清單中的每一個有效後置都建立了僞樣式規則，如‘%.p' 。參閱舊式的後置規則(suffix rule)。

10.5.6刪除隱含規則

透過定義新的具有相同目標和先決條件但不同命令的規則，您可以重載內建的隱含規則（或重載您自己定義的規則）。一旦定義新的規則，內建的規則就被代替。 新規則在隱含規則次序表中的位置由您編寫規則的地方決定。
透過定義新的具有相同目標和先決條件但不含命令的規則，您可以刪除內建的隱含規則。例如，下面的定義規則將刪除執行彙編編譯器的隱含規則：
%.o : %.s

10.6 定義最新類型的預設規則

您透過編寫不含先決條件的最終萬用樣式規則，您可以定義最新類型的預設規則。參閱萬用規則。這和其它規則基本一樣，特別之處在於它可以匹配任何目標。因此，這樣的規則的命令可用於所有沒有自己的命令的目標和先決條件，以及用於那些沒有其它隱含規則可以應用的目標和先決條件。
例如，在測試makefile時，您可能不關心源檔案是否含有真實數據，僅僅關心它們是否存在。那麼，您可以這樣做：

```
%::
        touch $@
```

這導致所有必需的源檔案（作爲先決條件）都自動建立。

您可以爲沒有規則的目標以及那些沒有具體指定命令的目標定義命令。要完成上述任務，您需要爲特殊目標.DEFAULT 編寫規則。這樣的規則可以在所有具體規則中用於沒有作爲目標出現以及不能使用隱含規則的先決條件。自然，如果您不編寫定義則沒有特殊目標.DEFAULT 的規則。

如果您使用特殊目標.DEFAULT 而不帶任何規則和命令:

.DEFAULT:
則以前為目標.DEFAULT定義的命令被清除。如此make的行為和您從來沒有定義目標.DEFAULT一樣。
如果您不需要一個目標從萬用規則和目標.DEFAULT 中得到命令，也不想為該目標執行任何命令，您可以在定義時使用空命令。參閱使用空命令。
您可以使用最新類型規則重載另外一個makefile檔案的一部分內容。參閱重載其它makefile檔案。

10.7 舊式的後置規則(suffix rule)

後置規則(suffix rule)是定義隱含規則的過時方法。後置規則(suffix rule)因為樣式規則更為普遍和簡潔而被廢棄。它們在GNU make中得到支援是為了和早期的makefile檔案兼容。它們分為單後置和雙後置規則(suffix rule)。
雙後置規則(suffix rule)被一對後置定義：目標後置和源檔案後置。它可以匹配任何檔案名以目標後置結尾的檔案。相應的隱含先決條件透過在檔案名中將目標後置替換為源檔案後置得到。一個目標和源檔案後置分別為‘.o’和‘.c’雙後置規則(suffix rule)相當於樣式規則`%.o : %.c'。

單後置規則(suffix rule)被單後置定義，該後置是源檔案的後置。它匹配任何檔案名，其相應的先決條件名是將檔案名添加源檔案後置得到。源檔案後置為‘.c’的單後置規則(suffix rule)相當於樣式規則‘% : %.c’。
透過比較規則目標和定義的已知後置清單識別後追規則。當make見到一個目標後置是已知後置的規則時，該規則被認為是一個單後置規則(suffix rule)。當make見到一個目標後置引入(include)兩個已知後置的規則時，該規則被認為是一個雙後置規則(suffix rule)。

例如，‘.o’和‘.c’都是預設清單中的已知後置。所以，如果您定義一個規則，其目標是‘.c.o’，則make認為是一個雙後置規則(suffix rule)，源檔案後置是‘.c’，目標後置是‘.o’。這裡有一個採用舊式的方法定義編譯C語言程式的規則：

```
.c.o:
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

後置規則(suffix rule)不能有任何屬於它們自己的先決條件。如果它們有先決條件，它們將不是作為後置規則(suffix rule)使用，而是以令人啼笑皆非的模式處理正常的檔案。例如，規則：

```
.c.o: foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

告訴從先決條件foo.h生成檔案名為‘.c.o’的檔案，並不是像樣式規則：

```
%.o: %.c foo.h
	$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@ $<
```

告訴從‘.c'檔案生成 ‘.o' 檔案‘.c'的方法：建立所有‘.o' 檔案使用該樣式規則，而且同時使用先決條件檔案‘foo.h'。
沒有命令的後置規則(suffix rule)也沒有意義。它們並不沒有命令的樣式規則那樣移去以前的規則（參閱刪除隱含規則）。他們僅僅簡單的在資料庫中加入後置或雙後置作為一個目標。
已知的後置是特殊目標‘.SUFFIXES’簡單的先決條件名。透過為特殊目標‘.SUFFIXES’編寫規則加入更多的先決條件，您可以添加您自己的已知後置。例如：

```
.SUFFIXES: .hack .win
```

把‘.hack' 和‘.win'添加到了後置清單中。
如果您希望排除預設的已知後置而不是僅僅的添加後置，那麼您可以為特殊目標‘.SUFFIXES’編寫沒有先決條件的規則。透過這種模式，可以完全排除特殊目標‘.SUFFIXES’存在的先決條件。接著您可以編寫另外一個規則添加您要添加的後置。例如，

```
.SUFFIXES:            # 刪除預設後置
.SUFFIXES: .c .o .h   # 定義自己的後置清單
```

標誌‘-r'或‘--no-builtin-rules'也能把預設的後置清單清空。
變數SUFFIXES在make讀入任何makefile檔案之前定義預設的後置清單。您可以使用特殊目標‘.SUFFIXES’改變後置清單，但這不能改變變數SUFFIXES的值。

10.8隱含規則**搜尋算法**

這裡是make為一個目標‘t’搜尋隱含規則的過程。這個過程用於任何沒有命令的**雙冒號規則(::)**，用於任何不含命令的

普通規則的目標，以及用於任何不是其它規則目標的先決條件。這個過程也能用於來自 隱含規則的先決條件遞迴該過程搜尋規則鏈。

在本算法中不提及任何後置規則(suffix rule)，因爲後置規則(suffix rule)在makefile檔案讀入時轉化爲了樣式規則。

對於個是‘archive(member)’的資料庫成員目標，下述算法重複兩次，第一次使用整個目標名‘t’，如果第一次執行沒有發現規則，則第二次使用‘(member)’作爲目標‘t’。

1、 在‘t’中分離出路徑(stem)部分，稱爲‘d’，剩下部分稱爲‘n’。例如如果‘t’是‘src/foo.o’，那麼‘d’是‘src/’；‘n’是‘foo.o’。
2、 建立所有目標名匹配‘t’和‘n’的樣式規則列表。如果目標樣式中含有斜線(/)，則匹配‘t’，否則，匹配‘n’。
3、 如果列表中有一個規則不是萬用規則，則從列表中刪除所有非最終萬用規則。
4、 將沒有命令的規則也從列表中移走。
5、 對每個列表中的樣式規則：
 1、 尋找stem‘s’，也就是和目標樣式中%匹配的‘t’或‘n’部分。
 2、 使用stem‘s’計算依賴名。如果目標樣式不引入(include)斜線(/)，則將‘d’添加在每個先決條件的前面。
 3、測試所有的依賴是否存在或能夠建立。（如果任何檔案在makefile中作爲目標或先決條件被提及，則我們說它應該存在。）如果所有先決條件存在或能夠建立，或沒有先決條件，則可使用該規則。
6、 如果到現在還沒有發現能使用的規則，進一步試。對每一個清單中的規則：
 1、 如果規則是最終規則，則忽略它，繼續下一條規則。
 2、 像上述一樣計算依賴名。
 3、 測試所有的依賴是否存在或能夠建立。
 4、 對於不存在的依賴，按照該算法遞迴查找是否能夠采用隱含規則建立。
 5、 如果所有依賴存在或能使用隱含規則建立，則應用該規則。
7、 如果沒有隱含規則，則如有用於目標‘.DEFAULT’規則，則應用該規則。在這種情況下，將目標‘.DEFAULT’的命令給與‘t’。

一旦找到可以應用的規則，對每一個匹配的目標樣式（無論是‘t’或‘n’）使用stem‘s’替換%，將得到的檔案名儲存起來直到執行命令更新目標檔案‘t’。在這些命令執行以後，把每一個儲存的檔案名放入資料庫，並且標誌已經更新，其時間戳和目標檔案‘t’一樣。
如果樣式規則的命令爲建立‘t’執行，自動變數將設定爲相應的目標和先決條件（參閱自動變數）。

11使用make更新資料庫檔案
資料庫檔案是引入(include)子檔案的檔案，這些子檔案有各自的檔案名，一般將它們稱爲成員；資料庫檔案和程式ar一塊被提及，它們的主要用途是作爲連接的例程庫。

11.1資料庫成員目標
獨立的檔案資料庫成員可以在make中用作目標或先決條件。按照下面的模式，您可以在資料庫檔案‘archive’中指定名爲‘member’的成員：

archive(member)

這種架構僅僅在目標和先決條件中使用，絕不能在命令中應用！絕大多數程式都不在命令中支援這個語法，而且也不能對資料庫成員直接操作。只有程式ar和那些爲操作資料庫檔案設計的程式才能這樣做。所以合法的更新資料庫成員的命令一定使用ar。例如，下述規則表明借助拷貝檔案‘hack.o’在檔案‘foolib’中建立成員‘hack.o’:

```
foolib(hack.o) : hack.o
        ar cr foolib hack.o
```

實際上，幾乎所有的資料庫成員目標是採用這種模式更新的，並且有一條隱含規則爲您專門更新資料庫成員目標。**注意：如果資料庫檔案沒有直接存在，程式ar的‘c’標誌是需要的。**

在相同的檔案中同時指定幾個成員，您可以在圓括號中一起寫出所有的成員名。例如：

foolib(hack.o kludge.o)

等同於：

foolib(hack.o) foolib(kludge.o)

您還可以在資料庫成員引用中使用shell類型的萬用字元。參閱在檔案名中使用萬用字元。例如，‘foolib(*.o)' 展開爲在檔案‘foolib’中所有存在以‘.o’結尾的成員。也許相當於：‘foolib(hack.o) foolib(kludge.o)'。

11.2 資料庫成員目標的隱含規則
對目標‘a(m)’表示名爲‘m’的成員在資料庫檔案‘a’中。

Make爲這種目標搜尋隱含規則時，是用它另外一個的特殊特點：make認爲匹配‘(m)’的隱含規則也同時匹配‘a(m)’。

該特點導致一個特殊的規則，它的目標是‘(%)’。該規則透過將檔案‘m’拷貝到檔案中更新目標‘a(m)’。例如，它透過將檔案‘bar.o’拷貝到檔案‘foo.a’中更新資料庫成員目標‘foo.a(bar.o)’。

如果該規則和其它規則組成鏈，功能十分強大。‘make "foo.a(bar.o)"'（注意使用雙引號是爲了保護圓括號可被shell解釋）即使沒有makefile檔案僅存在檔案‘bar.c’就可以保證以下命令執行：

```
cc -c bar.c -o bar.o
ar r foo.a bar.o
rm -f bar.o
```

這裡make假設檔案‘bar.o’是中間檔案。參閱隱含規則鏈。

諸如這樣的隱含規則是使用自動變數‘$%’編寫的，參閱自動變數。

資料庫成員名不能引入(include)路徑(stem)名，但是在makefile檔案中路徑(stem)名是有用的。如果您寫一個資料庫成員規則‘foo.a(dir/file.o)’，make將自動使用下述命令更新：

```
ar r foo.a dir/file.o
```

它的結果是拷貝檔案‘dir/file.o’進入名爲‘file.a’的檔案中。在完成這樣的任務時使用自動變數%D和%F。

11.2.1更新檔案的符號索引表
用作庫的資料庫檔案通常引入(include)一個名爲‘__.SYMDEF’ 特殊的成員，成員‘__.SYMDEF’引入(include)由所有其它成員定義的外部符號名的索引表。在您更新其它成員後，您必須更新成員‘__.SYMDEF’，從而使成員‘__.SYMDEF’可以合適的總結其它成員。要完成成員‘__.SYMDEF’的更新需要執行ranlib程式：

```
ranlib archivefile
```

正常情況下，您應該將該命令放到資料庫檔案的規則中，把所有資料庫檔案的成員作爲該規則的先決條件。例如：

```
libfoo.a: libfoo.a(x.o) libfoo.a(y.o) ...
        ranlib libfoo.a
```

上述程式的結果是更新資料庫成員‘x.o',‘y.o', 等等, 然後透過執行程式ranlib更新符號索引表表成員‘__.SYMDEF’。更新成員的規則這裡沒有列出，多數情況下，您可以省略它們，使用隱含規則把檔案拷貝到檔案中，具體描述見以前的內容。

使用GNU ar程式時這不是必要的，因爲它自動更新成員‘__.SYMDEF’。

11.3 使用檔案的危險

同時使用並行執行（-j開關，參閱並行執行）和檔案應該十分小心。如果多個命令同時對相同的資料庫檔案操作，它們相互不知道，有可能破壞檔案。將來的make版本可能針對該問題提供一個機製，即將所有操作相同資料庫檔案的命令串行化。但是現下，您必須在編寫您自己的makefile檔案時避免該問題，或者採用其它模式，或者不使用選項-j。

11.4 資料庫檔案的後置規則(suffix rule)

為處理資料庫檔案，您可以編寫一個特殊類型的後置規則(suffix rule)。關於所有後置的展開請參閱舊式的後置規則(suffix rule)。檔案後置規則(suffix rule)在GNU make中已被廢棄，因為用於檔案的樣式規則更加通用（參閱資料庫成員目標的隱含規則），但是為了和其它版本的make兼容，它們仍然被保留。
編寫用於檔案的後置規則(suffix rule)，您可以簡單的編寫一個用於目標後置'.a'的後置規則(suffix rule)即可。

例如，這裡有一個用於從C語言源檔案更新檔案庫的過時後置規則(suffix rule)：

```
.c.a:
	$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
	$(AR) r $@ $*.o
	$(RM) $*.o
```

這和下面的樣式規則工作完全一樣：

```
(%.o): %.c
	$(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $*.o
	$(AR) r $@ $*.o
	$(RM) $*.o
```

實際上，這僅僅是make看到一個以'.a'作為後置的後置規則(suffix rule)時，它所做的工作。任何雙後置規則(suffix rule)'.x.a' 被轉化為一個樣式規則，該樣式規則的目標樣式是'(%.o)'，先決條件格式是'%.x'.。
因為您可能要使用'.a' 作為一個檔案類型的後置，make也以正常模式轉換檔案後置規則(suffix rule)為樣式規則，參閱舊式的後置規則(suffix rule)。這樣一個雙後置規則(suffix rule)'.x.a' 產生兩個樣式規則：'(%.o): %.x' 和'%.a: %.x'.

12 GNU make的特點

這裡是GNU make的特點的總結，用於比較其它版本的make。我們以4.2 BSD 中的make的特點爲基準。如果您要編寫一個可移植的makefile檔案，您不要使用這裡列出的make的特點，也不要使用不相容性和失去的特點中列出的內容。

- 許多特點在System V 中的make也存在。
- 變數VPATH 以及它特殊的意義。參閱在目錄中搜尋先決條件。這個特點存在於System V 中的make，但沒有事實證明。4.3 BSD make也含有該特點（據說是模仿System V中變數VPATFH的特點）。
- 引入(include)其它makefile檔案。參閱引入(include)其它makefile檔案。允許使用一個指令引入(include)多個檔案是GNU的展開。
- 透過環境，變數可以讀入和通訊，參閱環境變數。
- 透過變數MAKEFLAGS 在遞迴make時可以傳遞選項。參閱和子make通訊選項。
- 在檔案引用中自動變數$% 設定爲成員名。參閱自動變數。
- 自動變數$@, $*, $<, $%, 和 $? 有變體形式如$(@F)和$(@D)。我們把此概念化，並使用它對自動變數$^ 進行了明顯展開。參閱自動變數。
- 變數引用。參閱變數引用基礎。
- 命令行選項‘-b'和‘-m'，接受和忽略。在System V make中，這些選項實際起作用。
- 即使指定選項‘-n',‘-q'或‘-t'，也能透過變數MAKE執行地歸呼叫make的命令。參閱遞迴make。
- 在後置規則(suffix rule)中支援後置‘.a'。參閱用於資料庫檔案的後置規則(suffix rule)。這個特點在GNU make中幾乎不用，因爲規則鏈更加通用的特點（參閱隱含規則鏈）允許一個樣式規則用於在檔案中安裝成員已經足夠（參閱用於資料庫成員目標的隱含規則）。

在命令中行排列和反斜線(\)-新行結合依舊保留，當命令列印時，它們出現的格式和它們在makefile檔案中基本一樣，不同之處是去掉了初始化空白。

- 下面的特點被各種不同版本的make吸收，但哪些版本吸收了哪些特點並不十釐清楚。
- 在樣式規則中使用‘%’。已經有幾個不同版本的make使用了該特點。我們不能確認是誰發明了它，但它發展很快。參閱定義與重新定義樣式規則。
- 規則鏈以及隱含中間檔案。這個特點首先由Stu Feldman 在它的make版本中實現，並用於AT&T 第八版Unix研究中。後來AT&T貝拉實驗室的Andrew Hume 在它的mk程式中應用（這裡稱爲“傳遞閉合”）。我們並不清楚是從他們那裡得到這個特點或是同時我們自己開發出來的。參閱隱含規則鏈。
- 自動變數引入(include)當前目標的所有先決條件的清單。我們一點也不知道是誰做的。參閱自動變數。自動變數$+是變數$^的簡單展開。
- "what if" 標誌(GNU make中的‘-W') 是Andrew Hume 在mk中發明的。參閱代替執行命令。
- 並行執行的概念在許多版本的make中存在，儘管System V 或BSD 並沒有實現。參閱執行命令。
- 使用格式替換改變變數引用來自於SunOS 4。參閱變數引用基礎。在GNU make中，這個功能在變換語法和SunOS 4兼容之前由函數patsubst提供。不知道誰是威權，因爲GNU make 使用函數 patsubst 在 SunOS 4 發布之前。
- 在命令行前面的‘+’字符有特殊重要的意義（參閱代替執行命令）。這是由IEEE Standard 1003.2-1992 (POSIX.2)定義的。
- 使用‘+=語法爲變數追加值來自於SunOS 4 make。參閱爲變數值附加文字(text)。
- 語法‘archive(mem1 mem2...)'在單一資料庫檔案中列舉多個成員來自於SunOS 4 make.。參閱資料庫成員目標。

-include指令包括makefile檔案，並且對於不存在的檔案也不產生錯誤。該特點with來自於SunOS 4 make。(但是SunOS 4 make 在單個指令中指定多個makefile檔案。) 該特點和SGI make 的sinclude 相同，

- 剩餘的特點是由GNU make發明的：
- 使用‘-v'或`--version'選項列印版本和拷貝權訊息。
- 使用‘-h' 或‘--help' 選項總結make的選項。
- 簡單展開型變數。參閱變數的兩特特色。
- 在遞迴make時，透過變數MAKE自動傳遞命令行變數。參閱遞迴make。
- 使用命令選項‘-C' 或‘--directory'改變路徑(stem)。參閱選項概要。
- 定義多行變數。參閱定義多行變數。
- 使用特殊目標.PHONY聲明假想(phony)目標。AT&T 貝拉實驗室Andrew Hume 使用不同的語法在它的mk程式中也實現了該功能。這似乎是並行的發現。參閱假想(phony)目標。
- 呼叫函數操作文字(text)。參閱用於轉換文字(text)的函數。
- 使用‘-o'或‘--old-file'選項假裝檔案是舊檔案。參閱避免重新編譯檔案。

- 條件執行。該特點已在不同版本make中已經實現很長時間了；它似乎是C與處理程式和類似的巨集語言的自然展開，而不是革命性的概念。參閱makefile檔案中的條件語句。
- 指定引入(include)的makefile檔案的搜尋路徑(stem)。參閱引入(include)其它makefile檔案。
- 使用環境變數指定額外的makefile檔案。參閱變數MAKEFILES。
- 從檔案名中去除前導斜線(/) `./'，因此，'./file' 和 'file' 是指同一個檔案。
- 使用特別搜尋方法搜尋形式如'-lname'的庫先決條件。參閱連接庫(Link Libraries)搜尋目錄。
- 允許後置規則(suffix rule)中的後置引入(include)任何字符（參閱舊式的後置規則(suffix rule)）。在其它版本的make中後置必須以'.'開始，並且不能引入(include)'/'字符。
- 包吹跟蹤當前make級別適用的變數MAKWFILES的值，參閱遞迴make。
- 將任何在命令行中給出的目標放入變數MAKECMDGOALS。參閱指定最終目標的參數。
- 指定靜態樣式規則。參閱靜態樣式規則。
- 提供選擇性vpath搜尋。參閱在目錄中搜尋先決條件。
- 提供可計算的變數引用。參閱變數引用基礎。
- 更新makefile檔案。參閱重建makefile檔案。System V make 中有非常非常有限的來自於該功能的形式，它用於為make檢查SCCS檔案。
- 各種新建的隱含規則。參閱隱含規則目錄。

內建變數`MAKE_VERSION' 給出make的版本號。

13 不相容性和失去的特點

- 其它版本的make程式也有部分特點在GNU make中沒有實現。POSIX.2 標準 (IEEE Standard 1003.2-1992)規定不需要這些特點。
- ‘file((entry))' 形式的目標代表一個資料庫檔案的成員file。選擇該成員不使用檔案名，而是透過一個定義連接符號enty的OBJ檔案。該特點沒有被GNU make 吸收因爲該非標準組件將爲make加入資料庫檔案符號表的內部知識。參閱更新檔案符號索引表。
- 在後置規則(suffix rule)中以字符‘~’結尾的後置在System V make中有特別的含義；它們指和檔案名中沒有‘~’的檔案通訊的SCCS 檔案。例如，後置規則(suffix rule)‘.c~.o'將從名爲‘s.n.c'的SCCS檔案中抽取檔案‘n.o'。爲了完全覆蓋，需要這種整系列的後置規則(suffix rule)，參閱舊式的後置規則(suffix rule)。在GNU make中，這種整系列的後置規則(suffix rule)由勇於從SCCS檔案抽取的兩個樣式規則掌管，它們可和通用的規則結合成規則鏈，參閱隱含規則鏈。

在System V make中, 字元串‘$$@'又奇特的含義，在含有多個規則的先決條件中，它代表正在處理的特殊目標。這在GNU make沒有定義，因爲字元串‘$$'代表一個平常的字符‘$'。使用靜態樣式規則可以實現該功能的一部分（參閱靜態樣式規則）。System V make 中的規則：

$(targets): $$@.o lib.a

在 GNU make 中可以用靜態樣式規則代替：

- $(targets): %: %.o lib.a
- 在System V 和 4.3 BSD make中， 透過VPATH搜尋（參閱爲先決條件搜尋目錄）發現的檔案，它們的檔案名改變後加入到命令字元串中。我們認爲使用自動變數更簡單明了，所以不引進該特點。
- 在一些Unix make中，自動變數$*出現下規則的先決條件中有令人驚奇的特殊特點：展開爲該規則的目標全名。我們不能明白Unix make 在心中對這是怎樣考慮的，它和正常的變數$*定義完全不同。

在一些Unix make中，隱含規則搜尋（參閱使用隱含規則）明顯是爲所有目標做的，而不僅僅爲那些沒有命令的目標。這意味著：

```
foo.o:
    cc -c foo.c
```

- 在Unix make 有直覺知道‘foo.o' 依靠‘foo.c'。我們認爲這樣的用法易導致混亂。Make中先決條件的屬性已經定義好（至少對於GNU make是這樣），再做這樣的事情不合規矩。
- GNU make不引入(include)任何編譯以及與處理EFL程式的隱含規則。如果我們聽說誰使用EFL，我們樂意把它們加入。

在 SVR4 make中，一條後置規則(suffix rule)可以不含命令，它的處理模式和它含有空命令的處理模式一樣（參閱使用空命令）。例如：

.c.a:

將重載內建的後置規則(suffix rule)‘.c.a'。我們覺得對沒有命令的規則簡單的爲目標添加先決條件更爲簡潔。上述例子和在GNU make中下例的行爲相同。

- .c.a: ;

一些版本的make 呼叫shell使用‘-e'標誌，而不是‘-k'標誌 (參閱測試程式編譯)。標誌‘-e'告訴shell 一旦程式執行返回非零狀態就立即退出。我們認爲根據每一命令行是否需要需要特殊處理直接寫入命令中更爲清楚。

14 makefile檔案慣例

本章描述爲GNU make編寫makefile檔案的慣例。使用Automake將幫助您按照這些慣例編寫makefile檔案。

14.1 makefile檔案的通用慣例

任何makefile檔案都應該引入(include)這行：

```
SHELL = /bin/sh
```

避免在系統中變數SHELL可能繼承環境中值的麻煩。（在GNU make中這從來不是問題。）

不同的make程式有不同的後置清單和隱含規則，這有可能造成混亂或錯誤的行爲。因此最好的辦法是設定後置清單，在該清單中，僅僅引入(include)您在特定makefile檔案中使用的後置。例如：

```
.SUFFIXES:
.SUFFIXES: .c .o
```

第一行清除了後置清單，第二行定義了在該makefile中可能被隱含規則使用的後置。

不要假設 ‘.' 是命令執行的路徑(stem)。當您在建立程式過程中，需要執行僅是您程式包中一部分的程式時，請確認如果該程式是要建立程式的一部分使用‘./’，如果該程式是源代碼中不變的部分使用‘$(srcdir)’。沒有這些前綴，僅僅在當前路徑(stem)下搜索。

建造目錄（build directory）‘./’和源代碼目錄(source directory)‘$(srcdir)’的區別是很重要的，因爲用戶可以在‘configure’中使用‘--srcdir’選項建造一個單獨的目錄。下面的規則：

```
foo.1 : foo.man sedscript
        sed -e sedscript foo.man > foo.1
```

如果建立的目錄不是源代碼目錄將失敗，因爲檔案‘foo.man’和‘sedscript’在源代碼目錄下。

在使用GNU make時，依靠變數‘VPATH’搜尋源檔案在單個從屬性檔案存在情況下可以很好地工作，因爲make中自動變數‘$<’中含有源檔案的存在路徑(stem)。（許多版本的make僅在隱含規則中設值變數‘$<’。）例如這樣的makefile檔案目標：

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c bar.c -o foo.o
```

將被替換爲：

```
foo.o : bar.c
        $(CC) -I. -I$(srcdir) $(CFLAGS) -c $< -o $@
```

這是爲了保證變數‘VPATH’正確的工作。目標含有多個先決條件時，使用名了的‘$(srcdir)’是最容易的保證該規則很好工作的方法。例如，以上例子中的目標‘foo.l’最好寫爲：

```
foo.1 : foo.man sedscript
        sed -e $(srcdir)/sedscript $(srcdir)/foo.man > $@
```

GNU的分類中通常引入(include)一些不是源檔案的檔案——例如，‘Info’檔案、從Autoconf, Automake, Bison 或 Flex中輸出的檔案等。這些檔案在源檔案目錄下，它們也應該在源檔案目錄下，不應該在建造目錄下。因此makefile規則應在源檔案目錄下更新它們。

然而，如果一個檔案沒有在分類中出現，makefile檔案不應把它們放到源檔案目錄下，因爲按照通常情況建立一個程式，不應該以任何模式更改源檔案目錄。

試圖建造的建立和安裝目標，至少（以及它們的子目標）可在並行的make中正確的工作。

14.2 makefile檔案的工具

編寫在shell sh中執行而不在csh中執行的makefile檔案命令(以及shell的腳本，例如‘configure’)，不要使用任何ksh或bash的特殊特點。

用於建立和安裝的‘configure’腳本和Makefile 規則除下面所列出工具外不應該直接使用其它的任何工具：

```
cat cmp cp diff echo egrep expr false grep install-info
ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true
```

壓縮程式gzip可在dist規則中使用。

堅持使用用於這些程式的通用選項，例如，不要使用‘mkdir -p'，它可能比較方便，但是其它大多數系統卻不支援它。

避免在makefile中創造符號連接是非常不錯的注意，因爲一些系統不支援這種做法。

用於建立和安裝的Makefile 規則可以使用編譯器以及相關的程式，但應該透過make變數使用它們，這樣可以方便用戶使用別的進行替換。這裡有按照我們的理念編寫一些程式：
ar bison cc flex install ld ldconfig lex
make makeinfo ranlib texi2dvi yacc
請使用下述make變數執行這些程式:
$(AR) $(BISON) $(CC) $(FLEX) $(INSTALL) $(LD) $(LDCONFIG) $(LEX)
$(MAKE) $(MAKEINFO) $(RANLIB) $(TEXI2DVI) $(YACC)
使用ranlib或ldconfig，您應該確定如果系統中不存在要使用的程式不會引起任何副作用。安排忽略這些命令產生的錯誤，並且列印訊息告訴用戶該命令執行失敗並不意味著存在問題。（Autoconf‘AC_PROG_RANLIB'巨集可在這方面幫助您。）如果您使用符號連接，對於不支援符號連接的系統您應該有一個低效率執行方案。
附加的工具也可透過make變數使用：
chgrp chmod chown mknod
它在makefile中（或腳本中），您知道引入(include)這些工具的特定系統中它都可以很好的工作。

14.3 指定命令的變數

Makefile檔案應該爲重載的特定命令、選項等提供變數。
特別在您執行大部分工具時都應該應用變數，如果您要使用程式Bison, 名爲BISON 的變數它的預設值設定爲：‘BISON = bison’，在您需要使用程式Bison時，您可以使用$(BISON)引用。
檔案管理器工具如ln, rm, mv等等，不必要使用這種模式引用，因爲用戶不可能使用別的程式替換它們。
每一個程式變數應該和用於向該程式提供選項的選項變數一起提供。在程式名變數後添加‘FLAGS'表示向該程式提供選項的選項變數--例如, BISONFLAGS。(名爲CFLAGS的變數向C編譯器提供選項, 名爲YFLAGS的變數向yacc提供選項，名爲LFLAGS的變數向lex提供選項等是這個規則例外，但因爲它們是標準所以我們保留它們。) 在任何進行預處理的編譯命令中使用變數CPPFLAGS ，在任何進行連接的編譯命令中使用變數LDFLAGS 和直接使用程式ld一樣。
對於C編譯器在編譯特定檔案時必須使用的選項，不應引入(include)在變數CFLAGS中，因爲用戶希望他們能夠自由的指定變數CFLAGS。 要獨立於變數CFLAGS安排向C編譯器傳遞這些必要的選項, 可以將這些選項寫入編譯命令行中或隱含規則的定義中，如下例：

```
CFLAGS = -g
ALL_CFLAGS = -I. $(CFLAGS)
.c.o:
	$(CC) -c $(CPPFLAGS) $(ALL_CFLAGS) $<
```

變數CFLAGS中包括選項‘-g’，因爲它對於一些編譯並不是必需的，您可以認爲它是預設推薦的選項。如果數據包建立使用GCC作爲編譯器，則變數CFLAGS中包括選項‘-o’,而且以它爲預設值。
將變數CFLAGS放到編譯命令的最後，在引入(include)編譯選項其它變數的後邊，因此用戶可以使用變數CFLAGS對其它變數進行重載。

每次呼叫C編譯器都用到變數CFLAGS ，無論進行編譯或連接都一樣。
任何Makefile檔案都定義變數INSTALL，變數INSTALL是將檔案安裝到系統中的基本命令。
任何Makefile檔案都定義變數INSTALL_PROGRAM 和INSTALL_DATA，(它們的預設值都是$(INSTALL)。) 在實際安裝程式時，不論可執行程式或非可執行程式，一般都使用它們作爲命令。下面是使用這些變數的例子：
$(INSTALL_PROGRAM) foo $(bindir)/foo
$(INSTALL_DATA) libfoo.a $(libdir)/libfoo.a

您可以隨意將變數DESTDIR預先設定爲目標檔案名。這樣做允許安裝程式建立隨後在實際目標檔案系統中安裝檔案的快照。不要再makefile檔案中設定變數DESTDIR，也不要引入(include)在安裝檔案中。用變數DERSTDIR改變上述例子：
$(INSTALL_PROGRAM) foo $(DESTDIR)$(bindir)/foo
$(INSTALL_DATA) libfoo.a $(DESTDIR)$(libdir)/libfoo.a
在安裝命令中一般使用檔案名而不是路徑(stem)名作爲第二個參數。對每一個安裝檔案都使用單獨的命令。

14.4安裝路徑(stem)變數

安裝目錄經常以變數命名，所以在非標準地方安裝也很容易，這些變數的標準名字將在下面介紹。安裝目錄依據標準檔案系統佈局，變數的變體已經在SVR4, 4.4BSD, Linux, Ultrix v4, 以及其它現代作業系統中使用。
以下兩個變數設定安裝檔案的根目錄，所有的其它安裝目錄都是它們其中一個的子目錄，沒有任何檔案可以直接安裝

在這兩個根目錄下。

`prefix'
前綴是用於構造以下列舉變數的預設值。變數prefix預設值是‘/usr/local'。建造完整的GNU系統時，變數prefix的預設值是空值，‘/usr' 是符號連接符‘/'。(如果您使用Autoconf，應將它寫為‘@prefix@'。)使用不同於建立程式時變數prefix的值執行‘make install'，不會重新編譯程式。

`exec_prefix'
前綴是用於構造以下列舉變數的預設值。變數exec_prefix預設值是$(prefix). (如果您使用Autoconf，應將它寫為`@exec_prefix@'。) 一般情況下，變數$(exec_prefix) 用於存放引入(include)機器特定檔案的目錄，（例如可執行檔案和例程庫），變數$(prefix) 直接存放其它目錄。使用不同於建立程式時變數exec_prefix的值執行‘make install'，不會重新編譯程式。

可執行程式安裝在以下目錄中：
`bindir'
這個目錄下用於安裝用戶可以執行的可執行程式。其正常的值是‘/usr/local/bin'，但是使用時應將它寫為‘$(exec_prefix)/bin'。 (如果您使用Autoconf, 應將它寫為‘@bindir@'。)

`sbindir'
這個目錄下用於安裝從shell中呼叫執行的可執行程式。它僅僅對系統管理員有作用。它的正常的值是‘/usr/local/sbin'，但是使用時應將它寫為‘$(exec_prefix)/sbin'。 (如果您使用Autoconf, 應將它寫為‘@sbindir@'。)

`libexecdir'
這個目錄下用於安裝其它程式呼叫的可執行程式。其正常的值是‘/usr/local/libexec'，但是使用時應將它寫為‘$(exec_prefix)/libexec'。(如果您使用Autoconf, 應將它寫為‘@libexecdir@'。)

- 程式執行時使用的數據檔案可分為兩類：
- 程式可以正常更改的檔案和不能正常更改的檔案（雖然用戶可以編輯其中的一部分檔案）。

體系架構無關檔案，指這些檔案可被所有機器共享；體系相關檔案，指僅僅可以被相同類型機器、作業系統共享的檔案；其它是永遠不能被兩個機器共享的檔案。
這可產生六種不同的可能性。我們極力反對使用體系相關的檔案，當然OBJ檔案和庫檔案除外。使用其它體系無關的數據檔案更加簡潔，並且，這樣做也不是很難。
所以，這裡有 Makefile變數用於指定路徑(stem)：

`datadir'
這個目錄下用於安裝只讀型體系無關數據檔案。其正常的值是‘/usr/local/share'，但是使用時應將它寫為‘$(prefix)/share'。(如果您使用Autoconf, 應將它寫為‘@datadir@'。) 作為例外，參閱下述的變數‘$(infodir)'和‘$(includedir)'。

`sysconfdir'
這個目錄下用於安裝從屬於單個機器的只讀數據檔案，這些檔案是：用於配置主機的檔案。郵件服務、網路配置檔案，‘/etc/passwd'，等等都屬於這裡的檔案。所有該目錄下的檔案都是平常的ASCII文字(text)檔案。其正常的值是‘/usr/local/etc', 但是使用時應將它寫為‘$(prefix)/etc'. (如果您使用Autoconf, 應將它寫為‘@sysconfdir@'.) 不要在這裡安裝可執行檔案（它們可能屬於‘$(libexecdir)'或‘$(sbindir)'）。也不要在這裡安裝那些在使用時要更改的檔案（這些程式用於改變系統拒絕的配置）。它們可能屬於‘$(localstatedir)'。

`sharedstatedir'
這個目錄下用於安裝程式執行中要發生變化的體系無關數據檔案。其正常的值是‘/usr/local/com'，但是使用時應將它寫為‘$(prefix)/com'。 (如果您使用Autoconf, 應將它寫為‘@sharedstatedir@'。)

`localstatedir'
這個目錄下用於安裝程式執行中要發生變化的數據檔案。但他們屬於特定的機器。用戶永遠不需要在該目錄下更改檔案配置程式包選項；將這些配置訊息放在分離的檔案中，這些檔案將放入‘$(datadir)'或‘$(sysconfdir)'中，‘$(localstatedir)'正常的值是‘/usr/local/var'，但是使用時應將它寫為‘$(prefix)/var'。(如果您使用Autoconf, 應將它寫為‘

@localstatedir@'。)

`libdir'
這個目錄下用於存放OBJ檔案和庫的OBJ代碼。不要在這裡安裝可執行檔案，它們可能應屬於‘$(libexecdir)'。變數libdir正常的值是‘/usr/local/lib'，但是使用時應將它寫爲‘$(exec_prefix)/lib'。(如果您使用Autoconf, 應將它寫爲‘@libdir@'。)

`infodir'
這個目錄下用於安裝套裝軟件的 Info 檔案。預設情況下其值是‘/usr/local/info'，但是使用時應將它寫爲‘$(prefix)/info'. (如果您使用Autoconf, 應將它寫爲‘@infodir@'.)

`lispdir'
這個目錄下用於安裝套裝軟件的Emacs Lisp 檔案。預設情況下其值是‘/usr/local/share/emacs/site-lisp'，但是使用時應將它寫爲‘$(prefix)/share/emacs/site-lisp'。如果您使用Autoconf, 應將它寫爲‘@lispdir@'。爲了保證‘@lispdir@'工作，您需要將以下幾行加入到您的‘configure.in'檔案中：
lispdir='${datadir}/emacs/site-lisp'
AC_SUBST(lispdir)

`includedir'
這個目錄下用於安裝用戶程式中C‘#include'預處理指令引入(include)的頭檔案。其正常的值是‘/usr/local/include'，但是使用時應將它寫爲‘$(prefix)/include'。(如果您使用Autoconf, 應將它寫爲‘@includedir@'。) 除GCC外的大多數編譯器不在目錄‘/usr/local/include'搜尋頭檔案，因此這種安裝模式僅僅適用於GCC。有時，這也不是問題，因爲一部分庫檔案僅僅依靠GCC才能工作。但也有一部分庫檔案依靠其他編譯器，它們將它們的頭檔案安裝到兩個地方，一個由變數 includedir 指定，另一個由變數oldincludedir指定。

`oldincludedir'
這個目錄下用於安裝‘#include'的頭檔案，這些頭檔案用於除GCC外的其它C語言編譯器。其正常的值是‘/usr/include'。(如果您使用Autoconf, 應將它寫爲‘@oldincludedir@'。) Makefile命令變數oldincludedir 的值是否爲空，如果是空值，它們不在試圖使用它，它們還刪除第二次安裝的頭檔案。一個套裝軟件在該目錄下替換已經存在的頭檔案，除非頭檔案來源於同一個套裝軟件。例如，如果您的套裝軟件Foo 提供一個頭檔案‘foo.h'，則它在變數oldincludedir指定的目錄下安裝的條件是 (1) 這裡沒有投檔案‘foo.h' 或 (2) 來源於套裝軟件Foo的頭檔案‘foo.h'已經在該目錄下存在。要檢查頭檔案‘foo.h'是否來自於套裝軟件Foo，將一個magic字元串放到檔案中--作爲命令的一部分--然後使用正則規則（grep）查找該字元串。

Unix風格的幫助檔案安裝在以下目錄中：
`mandir'
安裝該套裝軟件的頂層幫助（如果有）目錄。其正常的值是‘/usr/local/man'，但是使用時應將它寫爲‘$(prefix)/man'。(如果您使用Autoconf, 應將它寫爲‘@mandir@'。)
`man1dir'
這個目錄下用於安裝第一層幫助。其正常的值是‘$(mandir)/man1'。
`man2dir'
這個目錄下用於安裝第一層幫助。其正常的值是‘$(mandir)/man2'。
`...'
不要將任何GNU 軟體的主要文檔作爲幫助頁。應該編寫使用手冊。幫助頁僅僅是爲了人們在Unix上方便執行GNU軟體，它是附屬的執行程式。
`manext'
檔案名表示對已安裝的幫助頁的展開。它引入(include)一定的週期，後跟適當的數字，正常爲‘1’。
`man1ext'
檔案名表示對已安裝的幫助頁第一部分的展開。
`man2ext'
檔案名表示對已安裝的幫助頁第二部分的展開。
`...'
使用這些檔案名代替`manext'。如果該套裝軟件的幫助頁需要安裝使用手冊的多個章節。
最後您應該設定一下變數：

`srcdir'
這個目錄下用於安裝要編譯的原檔案。該變數正常的值由shell腳本configure插入。(如果您使用Autoconf, 應將它寫爲‘srcdir = @srcdir@'.)

例如：
用於安裝路徑(stem)的普通前綴。
注意：該路經在您開始安裝時必須存在
prefix = /usr/local
exec_prefix = $(prefix)
這裡放置`gcc'命令呼叫的可執行檔案。
bindir = $(exec_prefix)/bin
這裡放置編譯器使用的目錄。
libexecdir = $(exec_prefix)/libexec
#這裡放置Info檔案。
infodir = $(prefix)/info
如果您的程式要在標準用戶指定的目錄中安裝大量的檔案，將該程式的檔案放入到特意指定的子目錄中是很有必要的。如果您要這樣做，您應該寫安裝規則建立這些子目錄。
不要期望用戶在上述列舉的變數值中包括這些子目錄，對於安裝目錄使用一套變數名的辦法使用戶能夠對於不同的GNU套裝軟件指定精確的值，爲了使這種做法有用，所有的套裝軟件必須設計爲當用戶使用時它們能夠聰明的工作。

14.5用戶標準目標
所有的GNU程式中，在makefile中都有下列目標：

`all'
編譯整個程式。這應該是預設的目標。該目標不必重建文檔檔案，Info檔案已正常情況下應該包括在各個發布的檔案中，DVI檔案只有在明確請求情況下才重建。預設時，make規則編譯和連接使用選項‘-g'，所以程式調試只是像徵性的。對於不介意缺少幫助的用戶如果他們希望將可執行程式和幫助分開，可以從中剝離出可執行程式。

`install'
編譯程式並將可執行程式、庫檔案等拷貝到爲實際使用保留的檔案名下。如果是證實程式是否適合安裝的簡單測試，則該目標應該執行該測試程式。不要在安裝時剝離可執行程式，魔鬼很可能關心那些使用install-strip目標來剝離可執行程式的人。如果這是可行的，編寫的install目標規則不應該更改程式建造的目錄下的任何東西，僅僅提供‘make all'一切都能完成。這是爲了方便用戶命名和在其它系統安裝建造程式，如果要安裝程式的目錄不存在，該命令應能建立所有這些目錄，這包括變數prefix和exec_prefix特別指定的目錄和所有必要的子目錄。完成該任務的方法是借助下面描述的目標installdirs。在所有安裝幫助頁的命令前使用‘-'使make 能夠忽略這些命令產生的錯誤，這可以確保在沒有Unix幫助頁的系統上安裝該套裝軟件時能夠順利進行。安裝Info檔案的方法是使用變數$(INSTALL_DATA)將Info檔案拷貝到變數‘$(infodir)'中（參閱指定命令的變數），如果 install-info程式存在則執行它。 install-info是一個編輯Info‘dir'檔案的程式，它可以爲Info檔案添加或更新選單；它是Texinfo套裝軟件的一部分。這裡有一個安裝Info檔案的例子：

```
$(DESTDIR)$(infodir)/foo.info: foo.info
        $(POST_INSTALL)
# 可能在‘.’下有新的檔案，在srcdir中沒有。
        -if test -f foo.info; then d=.; \
         else d=$(srcdir); fi; \
        $(INSTALL_DATA) $$d/foo.info $(DESTDIR)$@; \
#如果 install-info程式存在則執行它。
# 使用‘if'代替在命令行前的‘-'
# 這樣，我們可以注意到執行install-info產生的真正錯誤。
# 我們使用‘$(SHELL) -c' 是因爲在一些shell中
# 遇到未知的命令不會執行失敗。
        if $(SHELL) -c 'install-info --version' \
           >/dev/null 2>&1; then \
          install-info --dir-file=$(DESTDIR)$(infodir)/dir \
                     $(DESTDIR)$(infodir)/foo.info; \
```

else true; fi

在編寫install目標時，您必須把所有的命令歸位三類：正常的命令、 安裝前命令和安裝後命令。參閱安裝命令分類。

`uninstall'
刪除所有安裝的檔案—有‘install’ 目標拷貝的檔案。該規則不應更改編譯產生的目錄，僅僅刪除安裝檔案的目錄。反安裝命令像安裝命令一樣分為三類，參閱安裝命令分類。

`install-strip'
和目標install類似，但在安裝時僅僅剝離出可執行檔案。 在許多情況下，該目標的定義非常簡單：

```
install-strip:
        $(MAKE) INSTALL_PROGRAM='$(INSTALL_PROGRAM) -s' \
                install
```

正常情況下我們不推薦剝離可執行程式進行安裝，只有您確信這些程式不會產生問題時才能這樣。剝離安裝一個實際執行的可執行檔案同時儲存那些在這種場合存在BUG的可執行檔案是顯而易見的。

`clean'
刪除所有當前目錄下的檔案，這些檔案正常情況下是那些‘建立程式’ 建立的檔案。不要刪除那些記錄配置的檔案，同時也應該保留那些‘建立程式’ 能夠修改的檔案，正常情況下要刪除的那些檔案不包括這些檔案，因為發布檔案是和這些檔案一起建立的。如果‘.dvi'檔案不是檔案發布檔案的一部分，則使用目標‘clean’ 將同時刪除‘.dvi'檔案。

`distclean'
刪除所有當前目錄下的檔案，這些檔案正常情況下是那些‘建立程式’ 或‘配置程式’ 建立的檔案。如果您不解包源程式，‘建立程式’ 不會建立任何其它檔案，‘make distclean'將僅在檔案發布檔案中留下原有的檔案。

`mostlyclean'
和目標‘clean'類似，但是避免刪除人們正常情況下不編譯的檔案。例如，用於GCC的目標‘mostlyclean不刪除檔案‘libgcc.a'，因為在絕大多數情況下它都不需要重新編譯。

`maintainer-clean'
幾乎在當前目錄下刪除所有能夠使用該makefile檔案可以重建的檔案。使用該目標刪除的檔案包括使用目標distclean,刪除的檔案加上從Bison產生的C語言源檔案和標誌清單、Info檔案等等。我們說“幾乎所有檔案”的原因是執行命令‘make maintainer-clean'不刪除腳本‘configure'，即使腳本‘configure'可以使用Makefile檔案建立。更確切地說，執行‘make maintainer-clean'不刪除為了執行腳本‘configure'以及開始建立程式的涉及的所有檔案。這是執行‘make maintainer-clean' 刪除所有能夠重新建立檔案時唯一不能刪除的一類檔案。目標‘maintainer-clean'由該套裝軟件的養護程式使用，不能被普通用戶使用。您可以使用特殊的工具重建被目標‘make maintainer-clean'刪除的檔案。因為這些檔案正常情況下引入(include)在發布的檔案中，我們並不關心它們是否容易重建。如果您發現您需要對全部發布的檔案重新解包，您不能責怪我們。要幫助make 的用戶意識到這一點，用於目標maintainer-clean 應以以下兩行為開始：
@echo‘該命令僅僅用於養護程式；’
@echo‘它刪除的所有檔案都能使用特殊工具重建。’

`TAGS'
更新該程式的標誌表。

`info'
產生必要的Info檔案。最好的方法是編寫像下面規則：

```
info: foo.info

foo.info: foo.texi chap1.texi chap2.texi
        $(MAKEINFO) $(srcdir)/foo.texi
```

您必須在makefile檔案中定以變數MAKEINFO。它將執行makeinfo程式，該程式是發布程式中Texinfo的一部分。正常情況下，一個GNU發布程式和Info檔案一起建立，這意味著Info檔案存在於源檔案的目錄下。當用戶建造一個套裝軟件，一般情況下，make不更新Info檔案，因為它們已經更新到最新了。

`dvi'
建立DVI檔案用於更新Texinfo文檔。例如：
dvi: foo.dvi

foo.dvi: foo.texi chap1.texi chap2.texi
        $(TEXI2DVI) $(srcdir)/foo.texi
您必須在makefile檔案中定義變數TEXI2DVI。它將執行程式texi2dvi，該程式是發布的Texinfo一部分。要么僅僅編寫依靠檔案，要么允許GNU make提供命令，二者必選其一。

`dist'
爲程式建立一個tar檔案。建立tar檔案可以將其中的檔案名以子目錄名開始，這些子目錄名可以是用於發布的套裝軟件名。另外，這些檔案名中也可以引入(include)版本號，例如，發布的GCC 1.40版的tar檔案解包的子目錄爲‘gcc-1.40'。最方便的方法是建立合適的子目錄名，如使用in或cp等作爲子目錄，在它們的下面安裝適當的檔案，然後把tar檔案解包到這些子目錄中。使用gzip壓縮這些tar檔案，例如，實際的GCC 1.40版的發布檔案叫‘gcc-1.40.tar.gz'。目標dist明顯的依靠所有的發布檔案中不是源檔案的檔案，所以你應確保發布中的這些檔案已經更新。參閱GNU標準編碼中建立發布檔案。

`check'
執行自我檢查。用戶應該在執行測試之前，應該先建立程式，但不必安裝這些程式；您應該編寫一個自我測試程式，在程式已建立但沒有安裝時執行。
以下目標建議使用習慣名，對於各種程式它們很有用：

installcheck
執行自我檢查。用戶應該在執行測試之前，應該先建立、安裝這些程式。您不因該假設‘$(bindir)'在搜尋路徑(stem)中。

installdirs
添加名爲‘installdirs'目標對於建立檔案要安裝的目錄以及它們的父目錄十分有用。腳本‘mkinstalldirs'是專爲這樣處理方便而編寫的；您可以在Texinfo套裝軟件中找到它，您可以像這樣使用規則：

```
# 確保所有安裝目錄(例如 $(bindir))
# 都實際存在，如果沒有則建立它們。
installdirs: mkinstalldirs
        $(srcdir)/mkinstalldirs $(bindir) $(datadir) \
                                $(libdir) $(infodir) \
                                $(mandir)
```

該規則並不更改編譯時建立的目錄，它僅僅建立安裝目錄。

14.6 安裝命令分類

編寫已安裝目標，您必須將所有命令分爲三類：正常的命令、安裝前命令和安裝後命令。
正常情況下，命令把檔案移動到合適的地方，並設定它們的模式。它們不會改變任何檔案，僅僅把它們從套裝軟件中完整地抽取出來。
安裝前命令和安裝後命令可能更改一些檔案，如，它們編輯配置檔案後資料庫檔案。
安裝前命令在正常命令之前執行，安裝後命令在正常命令執行後執行。
安裝後命令最普通的用途是執行install-info程式。 這種工作不能由正常命令完成，因爲它更改了一個檔案（Info 目錄），該檔案不能全部、單獨從套裝軟件中安裝。它是一個安裝後命令，因爲它需要在正常命令安裝套裝軟件中的Info檔案後才能執行。

許多程式不需要安裝前命令，但是我們提供這個特點，以便在需要時可以使用。
要將安裝規則的命令分爲這三類，應在命令中間插入category lines（分類行）。 分類行指定了下面敘述的命令的類別。分類行引入(include)一個Tab、一個特殊的make變數引用，以及行結尾的隨機註釋。您可以使用三個變數，每一個變數對應一個類別；變數名指定了類別。分類行不能出現下普通的執行檔案中，因爲這些make變數被由正常的定義（您也不應在makefile檔案中定義）。

這裡有三種分類行，後面的註釋解釋了它的含義：

```
$(PRE_INSTALL)     # 以下是安裝前命令
$(POST_INSTALL)    # 以下是安裝後命令
$(NORMAL_INSTALL)  # 以下是正常命令
```

如果在安裝規則開始您沒有使用分類行，則在第一個分類行出現之前的所有命令都是正常命令。如果您沒有使用任何分類行，則所有命令都是正常命令。

這是反安裝的分類行

```
$(PRE_UNINSTALL)     #以下是反安裝前命令
$(POST_UNINSTALL)    #以下是反安裝後命令
$(NORMAL_UNINSTALL)  #以下是正常命令
```

反安裝前命令的典型用法是從Info目錄刪除全部內容。

如果目標install或 uninstall 有先決條件作爲安裝程式的次程序，那麼您應該使用分類行先啓動每一個先決條件的命令，再使用分類行啓動主目標的命令。無論哪一個先決條件實際執行，這種模式都能保證每一條命令都放置到了正確的分類中。

安裝前命令和安裝後命令除了對於下述命令外，不能執行其它程式：

```
basename bash cat chgrp chmod chown cmp cp dd diff echo
egrep expand expr false fgrep find getopt grep gunzip gzip
hostname install install-info kill ldconfig ln ls md5sum
mkdir mkfifo mknod mv printenv pwd rm rmdir sed sort tee
test touch true uname xargs yes
```

按照這種模式區分命令的原因是爲了建立二進制套裝軟件。典型的二進制套裝軟件包括所有可執行檔案、必須安裝的其它檔案以及它自己的安裝檔案——所以二進制套裝軟件不需要執行任何正常命令。但是安裝二進制套裝軟件需要執行安裝前命令和安裝後命令。

建造二進制套裝軟件的程式透過抽取安裝前命令和安裝後命令工作。這裡有一個抽取安裝前命令的方法：

```
make -n install -o all \
    PRE_INSTALL=pre-install \
    POST_INSTALL=post-install \
    NORMAL_INSTALL=normal-install \
  | gawk -f pre-install.awk
```

這個檔案‘pre-install.awk'可能包括：

```
$0 ~ /^\t[ \t]*(normal_install|post_install)[ \t]*$/ {on = 0}
on {print $0}
$0 ~ /^\t[ \t]*pre_install[ \t]*$/ {on = 1}
```

安裝前命令的結果檔案是像安裝二進制套裝軟件的一部分shell腳本一樣執行。

15 快速參考

這是對指令、文字(text)操作函數以及GNU make能夠理解的變數等的匯總。對於其他方面的總結參閱特殊的內建目標名，隱含規則目錄，選項概要。

這裡是GNU make是別的指令的總結：

define variable
endef
定義多行遞迴展開型變數。參閱定義固定次序的命令。

ifdef variable
ifndef variable
ifeq (a,b)
ifeq "a" "b"
ifeq 'a' 'b'
ifneq (a,b)
ifneq "a" "b"
ifneq 'a' 'b'
else
endif
makefile檔案中的條件展開，參閱makefile檔案中的條件語句。

include file
-include file
sinclude file
引入(include)其它makefile檔案，參閱引入(include)其它makefile檔案。

override variable = value
override variable := value
override variable += value
override variable ?= value
override define variable
endef
定義變數、對以前的定義重載、以及對在命令行中定義的變數重載。參閱撤銷(override)指令。

export
告訴make預設向子過程輸出所有變數，參閱與子make通訊的變數。

export variable
export variable = value
export variable := value
export variable += value
export variable ?= value
unexport variable
告訴make是否向子過程輸出一個特殊的變數。參業與子make通訊的變數。

vpath pattern path
製定搜尋匹配‘%’樣式的檔案的路徑(stem)。參閱vpath指令。

vpath pattern
去除以前為‘pattern’指定的所有搜尋路徑(stem)。

vpath
去除以前用vpath指令指定的所有搜尋路徑(stem)。

這裡是操作文字(text)函數的總結，參閱文字(text)轉換函數：

$(subst from,to,text)
在‘text’中用‘to’代替‘from’，參閱字元串替換與分析函數。

$(patsubst pattern,replacement,text)
在‘text’中用‘replacement’代替匹配‘pattern’字，參閱字元串替換與分析函數。

$(strip string)
從字元串中移去多餘的空格。參閱字元串替換與分析函數。

$(findstring find,text)
確定‘find’在‘text’中的位置。參閱字元串替換與分析函數。

$(filter pattern...,*text)*
在‘text’中選擇匹配‘pattern’的字。參閱字元串替換與分析函數。

$(filter-out pattern...,*text)*
在‘text’中選擇不匹配‘pattern’的字。參閱字元串替換與分析函數。

$(sort list)
將‘list’中的字按字母順序排序，並刪除重複的字。參閱字元串替換與分析函數。

$(dir names...)
從檔案名中抽取路徑(stem)名。參閱檔案名函數。

$(notdir names...)
從檔案名中抽取路徑(stem)部分。參閱檔案名函數。

$(suffix names...)
從檔案名中抽取非路徑(stem)部分。參閱檔案名函數。

$(basename names...)
從檔案名中抽取基本檔案名。參閱檔案名函數。

$(addsuffix suffix,names...)
為‘names’中的每個字添加後置。參閱檔案名函數。

$(addprefix prefix,names...)
為‘names’中的每個字添加前綴。參閱檔案名函數。

$(join list1,list2)
連接兩個並行的字清單。參閱檔案名函數。

$(word n,text)
從‘text’中抽取第n個字。參閱檔案名函數。

$(words text)
計算‘text’中字的數目。參閱檔案名函數。

$(wordlist s,e,text)
返回‘text’中s到e之間的字。參閱檔案名函數。

$(firstword names...)

在‘names…’中的第一個字。參閱檔案名函數。

$(wildcard pattern...)
尋找匹配shell檔案名樣式的檔案名。參閱wildcard函數。

$(error text...)
該函數執行時，make產生訊息爲‘text’的致命錯誤。參閱控制make的函數。

$(warning text...)
該函數執行時，make產生訊息爲‘text’的警告。參閱控制make的函數。

$(shell command)
執行shell命令並返回它的輸出。參閱函數shell。

$(origin variable)
返回make變數‘variable’的定義訊息。參閱函數origin。

$(foreach var,words,text)
將清單清單words中的每一個字對應後接var中的每一個字，將結果放在text中。參閱函數foreach。

$(call var,param,...)
使用對$(1), $(2)...對變數計算變數 var ，變數$(1), $(2)...分別代替參數 param 第一個、第二個…的值。參閱函數call。
這裡是對自動變數的總結，完整的描述參閱自動變數。

$@
目標檔案名。

$%
當目標是資料庫成員時，表示目標成員名。

$<
第一個先決條件名。

$?
所有比目標‘新’的先決條件的名字，名字之間用空格隔開。對於爲資料庫成員的先決條件，只能使用命名的成員。參閱使用make更新資料庫檔案。

$^
$+
所有先決條件的名字，名字之間用空格隔開。對於爲資料庫成員的先決條件，只能使用命名的成員。參閱使用make更新資料庫檔案。變數 $^ 省略了重複的先決條件，而變數 $+ 則按照原來次序保留重複項，

$*
和隱含規則匹配的徑(stem)。參閱樣式匹配。

$(@D)
$(@F)
變數$@.中的路徑(stem)部分和檔案名部分。

$(*D)
$(*F)
變數$*中的路徑(stem)部分和檔案名部分。

$(%D)

$(%F)
變數$%中的路徑(stem)部分和檔案名部分。

$(<D)
$(<F)
變數$<中的路徑(stem)部分和檔案名部分。

$(^D)
$(^F)
變數$^中的路徑(stem)部分和檔案名部分。

$(+D)
$(+F)
變數$+中的路徑(stem)部分和檔案名部分。

$(?D)
$(?F)
變數$?中的路徑(stem)部分和檔案名部分。

以下是GNU make使用變數：

MAKEFILES
每次呼叫make要讀入的Makefiles檔案。參閱變數MAKEFILES。

VPATH
對在當前目錄下不能找到的檔案搜索的路徑(stem)。參閱VPATH: 所有先決條件的搜尋路徑(stem)。

SHELL
系統預設命令解釋程式名，通常是`/bin/sh'。您可以在makefile檔案中設值變數SHELL改變執行程式使用的shell。參閱執行命令。

MAKESHELL
改變數僅用於MS-DOS，make使用的命令解釋程式名，該變數的值比變數SHELL的值優先。參閱執行命令。

MAKE
呼叫的make名。在命令行中使用該變數有特殊的意義。參閱變數MAKE的工作模式。

MAKELEVEL
遞迴的層數(子makes)。參閱與子make通訊的變數。

MAKEFLAGS
向make提供標誌。您可以在環境或makefile檔案中使用該變數設定標誌。參閱與子make通訊的變數。在命令行中不能直接使用該變數，應爲它的內容不能在shell中正確引用，但總是允許遞迴make時透過環境傳遞給子make。

MAKECMDGOALS
該目標是在命令行中提供給make的。設定該變數對make的行爲沒有任何影響。參閱特別目標的參數。
CURDIR
設定當前工作目錄的路徑(stem)名，參閱遞迴make。
SUFFIXES
在讀入任何makefile檔案之前的後置清單。
.LIBPATTERNS
定義make搜尋的庫檔案名，以及搜尋次序。參閱連接庫(Link Libraries)搜尋目錄。

16 make產生的錯誤

這裡是您可以看到的由make產生絕大多數普通錯誤清單，以及它們的含義和修正它們訊息。

有時make產生的錯誤不是致命的，如一般在命令腳本行前面存在前綴的情況下，和在命令行使用選向‘-k’的情況下產生的錯誤幾乎都不是致命錯誤。使用字元串***作前綴將產生致命的錯誤。

錯誤訊息前面都使用前綴，前綴的內容是產生錯誤的程式名或makefile檔案中存在錯誤的檔案名和引入(include)該錯誤的行的行號和。

在下述的錯誤清單中，省略了普通的前綴：

`[foo] Error NN'

`[foo] signal description'

這些錯誤並不是真的make的錯誤。它們意味著make呼叫的程式返回非零狀態值，錯誤碼（Error NN），這種情況make解釋為失敗，或非正常模式退出（一些類型信號），參閱命令錯誤。如果訊息中沒有附加***，則是子過程失敗，但在makefile檔案中的這條規則有特殊前綴，因此make忽略該錯誤。

`missing separator. Stop.'

`missing separator (did you mean TAB instead of 8 spaces?). Stop.'

這意味著make在讀取命令行時遇到不能理解的內容。GNU make 檢查各種分隔符(:, =, 字符TAB，等) 從而幫助確定它在命令行中遇到了什麼類型的錯誤。這意味著，make不能發現一個合法的分隔符。出現該訊息的最可能的原因是您（或許您的編輯器，絕大部分是ms-windows的編輯器）在命令行縮進使用了空格代替了字符Tab。這種情況下，make將使用上述的第二種形式產生錯誤訊息。**一定切記，任何命令行都以字符Tab開始，八個空格也不算數。**參閱規則的語法。

`commands commence before first target. Stop.'

`missing rule before commands. Stop.'

這意味著在makefile中似乎以命令行開始：以Tab字符開始，但不是一個合法的命令行（例如，一個變數的賦值）。任何命令行必須和一定的目標相聯繫。產生第二種的錯誤訊息是一行的第一個非空白字符為分號，make對此的解釋是您遺漏了規則中的"target: prerequisite" 部分，參閱規則的語法。

`No rule to make target `xxx'.'

`No rule to make target `xxx', needed by `yyy'.'

這意味著make決定必須建立一個目標，但卻不能在makefile檔案中發現任何用於建立該目標的指令，包括具體規則和隱含規則。如果您希望建立該目標，您需要另外為改目標編寫規則。其它關於該問題產生原因可能是makefile檔案是草稿（如檔案名錯）或破壞了源檔案樹（一個檔案不能按照計畫重建，僅僅由於一個先決條件的問題）。

`No targets specified and no makefile found. Stop.'

`No targets. Stop.'

前者意味著您沒有為命令行提供要建立的目標，make不能讀入任何makefile檔案。後者意味著一些makefile檔案被找到，但沒有引入(include)預設目標以及命令行等。GNU make在這種情況下無事可做。參閱指定makefile檔案的參數。

`Makefile `xxx' was not found.'

`Included makefile `xxx' was not found.'

在命令行中指定一個makefile檔案（前者）或引入(include)的makefile 檔案（後者）沒有找到。

`warning: overriding commands for target `xxx''

`warning: ignoring old commands for target `xxx''

GNU make允許命令在一個規則中只能對一個命令使用一次(**雙冒號規則(::)**除外)。如果您為一個目標指定一個命令，而該命令在目標定義是已經定義過，這種警告就會產生；第二個訊息表明後來設定的命令將改寫以前對該命令的設定。參閱具有多條規則的目標。

`Circular xxx <- yyy dependency dropped.'

這意味著make檢測到一個相互依靠一個循環：在跟蹤目標xxx的先決條件yyy 時發現，先決條件yyy的先決條件中一個又以xxx為先決條件。

`Recursive variable `xxx' references itself (eventually). Stop.'

這意味著您定義一個正常（遞迴性）make變數xxx，當它展開時，它將引用它自身。無論對於簡單展開型變數(:=)或追加定義（+=），這也都是不能允許的，參閱使用變數。

`Unterminated variable reference. Stop.'

這意味著您在變數引用或函數呼叫時忘記寫右括號。

`insufficient arguments to function `xxx'. Stop.'

這意味著您在呼叫函數是您密友提供需要數目的參數。關於函數參數的詳細描述參閱文字(text)轉換函數。

`missing target pattern. Stop.'

`multiple target patterns. Stop.'

`target pattern contains no `%'. Stop.'

這些錯誤訊息是畸形的靜態樣式規則引起的。第一條意味著在規則的目標部分沒有規則，第二條意味著在目標部分有多個規則，第三條意味著沒有引入(include)樣式符%。參閱靜態樣式規則語法。

`warning: -jN forced in submake: disabling jobserver mode.'

該條警告和下條警告是在make檢測到在能與子make通訊的系統中引入(include)並行處理的錯誤（參閱與子make通訊選項）。該警告訊息是如果遞迴一個make過程，而且還使用了‘-jn’選項（這裡n大於1）。這種情況很可能發生，例如，如果您設定環境變數MAKE爲‘make j2’。這種情況下，子make不能與其它make過程通訊， 而且還簡單假裝它由兩個任務。

`warning: jobserver unavailable: using -j1. Add `+' to parent make rule.'

爲了保證make過程之間通訊，父層make將傳遞訊息給子make。這可能導致問題，因爲子過程有可能不是實際的一個make，而父過程僅僅認爲子過程是一個make而將所有資訊傳遞給子過程。父過程是採用正常的算法決定這些的（參閱變數MAKE的工作模式）。如果makefile檔案構建了這樣的父過程，它並不知道子過程是否爲make，那麼，子過程將拒收那些沒有用的訊息。這種情況下，子過程就會產生該警告訊息，然後按照它內建的次序模式進行處理。

17 複雜的makfile檔案例子

這是一個用於GNU tar程式的makefile檔案；這是一個中等複雜的makefile檔案。
因為‘all’是第一個目標，所以它是預設目標。該makefile檔案一個有趣的地方是‘testpad.h'是由testpad程式建立的源檔案，而且該程式自身由‘testpad.c'編譯得到的。
如果您鍵入‘make'或`make all'，則make建立名為‘tar'可執行檔案, 提供遠程訪問卡帶的進程‘rmt'，和名為‘tar.info'的Info檔案。
如果您鍵入‘make install'，則make不但建立‘tar',‘rmt',和‘tar.info'，而且安裝它們。
如果您鍵入‘make clean', 則make刪除所有‘.o'檔案，以及‘tar',‘rmt',‘testpad',‘testpad.h',和‘core’檔案。
如果您鍵入‘make distclean', 則make不僅刪除‘make clean'刪除的所有檔案，而且包括檔案‘TAGS',‘Makefile', 和‘config.status' 檔案。(雖然不明顯，但該 makefile (和‘config.status')是用戶用configure程式產生的，該程式是由發布的tar檔案提供，但這裡不進行說明。)
如果您鍵入‘make realclean', 則make刪除‘make distclean '刪除的所有檔案，而且包括由‘tar.texinfo'產生的Info檔案。
另外，目標shar和dist創造了發布檔案的核心。

```
# 自動從makefile.in產生
# 用於GNU tar 程式的Unix Makefile
# Copyright (C) 1991 Free Software Foundation, Inc.

# 本程式是自由軟體；在遵照GNU條款的情況下
# 您可以重新發布它或更改它
# 普通公眾許可證 ...
...
...

SHELL = /bin/sh

#### 啟動系統配置部分 ####

srcdir = .

# 如果您使用gcc, 您應該在執行
# 和它一起建立的固定引入(include)的腳本程式以及
# 使用-traditional選項執行gcc中間選擇其一。
# 另外的ioctl呼叫在一些系統上不能正確編譯
CC = gcc -O
YACC = bison -y
INSTALL = /usr/local/bin/install -c
INSTALLDATA = /usr/local/bin/install -c -m 644

# 您應該在DEFS中添加的內容：
# -DSTDC_HEADERS          如果您有標準C的頭檔案和庫檔案。
# -DPOSIX                 如果您有POSIX.1的頭檔案和庫檔案。
# -DBSD42                 如果您有sys/dir.h (除非您使用-DPOSIX), sys/file.h,和st_blocks在`struct stat'中。
# -DUSG                   如果您有System V/ANSI C字元串和內存控制函數和頭檔案， sys/sysmacros.h,fcntl.h, getcwd, no
#                         valloc, 和 ndir.h (除非您使用-DDIRENT)。
# -DNO_MEMORY_H           如果USG或STDC_HEADERS 但是不包括memory.h.
# -DDIRENT                如果USG而且您又用dirent.h代替ndir.h。
# -DSIGTYPE=int           如果您的信號控制單元返回int,非void.
# -DNO_MTIO               如果您缺少sys/mtio.h (magtape ioctls).
# -DNO_REMOTE             如果您沒有一個遠程shell或rexec.
# -DUSE_REXEC             對遠程卡帶使用rexec操作代替分支rsh或remsh.
# -DVPRINTF_MISSING       如果您缺少函數vprintf(但是有_doprnt).
# -DDOPRNT_MISSING        如果您缺少函數 _doprnt.同樣需要定義 -DVPRINTF_MISSING.
# -DFTIME_MISSING         如果您缺少系統呼叫ftime
```

```
# -DSTRSTR_MISSING             如果您缺少函數strstr。
# -DVALLOC_MISSING             如果您缺少函數valloc。
# -DMKDIR_MISSING              如果您缺少系統呼叫mkdir和rmdir。
# -DRENAME_MISSING             如果您缺少系統呼叫rename。
# -DFTRUNCATE_MISSING          如果您缺少系統呼叫ftruncate。
# -DV7                         在Unix版本7 (沒有進行長期測試).
# -DEMUL_OPEN3                 如果您缺少3-參數版本的open, 並想透過您有的系統呼叫模擬它。
# -DNO_OPEN3                   如果您缺少3-參數版本的open並要禁止tar   k選項代替模擬open.
# -DXENIX                      如果您有sys/inode.h並需要它引入(include)94

DEFS =  -DSIGTYPE=int -DDIRENT -DSTRSTR_MISSING \
        -DVPRINTF_MISSING -DBSD42
# 設定為rtapelib.o ，除非使它為空時
# 您定義了NO_REMOTE,
RTAPELIB = rtapelib.o
LIBS =
DEF_AR_FILE = /dev/rmt8
DEFBLOCKING = 20

CDEBUG = -g
CFLAGS = $(CDEBUG) -I. -I$(srcdir) $(DEFS) \
        -DDEF_AR_FILE=\"$(DEF_AR_FILE)\" \
        -DDEFBLOCKING=$(DEFBLOCKING)
LDFLAGS = -g

prefix = /usr/local
# 每一個安裝程式的前綴。
#正常為空或`g'。
binprefix =

# 安裝tar的路徑(stem)
bindir = $(prefix)/bin

# 安裝info檔案的路徑(stem).
infodir = $(prefix)/info

#### 系統配置結束部分 ####

SRC1 =  tar.c create.c extract.c buffer.c \
        getoldopt.c update.c gnu.c mangle.c
SRC2 =  version.c list.c names.c diffarch.c \
        port.c wildmat.c getopt.c
SRC3 =  getopt1.c regex.c getdate.y
SRCS =  $(SRC1) $(SRC2) $(SRC3)
OBJ1 =  tar.o create.o extract.o buffer.o \
        getoldopt.o update.o gnu.o mangle.o
OBJ2 =  version.o list.o names.o diffarch.o \
        port.o wildmat.o getopt.o
OBJ3 =  getopt1.o regex.o getdate.o $(RTAPELIB)
OBJS =  $(OBJ1) $(OBJ2) $(OBJ3)
AUX =   README COPYING ChangeLog Makefile.in  \
        makefile.pc configure configure.in \
        tar.texinfo tar.info* texinfo.tex \
        tar.h port.h open3.h getopt.h regex.h \
        rmt.h rmt.c rtapelib.c alloca.c \
        msd_dir.h msd_dir.c tcexparg.c \
```

```
	level-0 level-1 backup-specs testpad.c

all:	tar rmt tar.info

tar:	$(OBJS)
	$(CC) $(LDFLAGS) -o $@ $(OBJS) $(LIBS)

rmt:	rmt.c
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ rmt.c

tar.info: tar.texinfo
	makeinfo tar.texinfo

install: all
	$(INSTALL) tar $(bindir)/$(binprefix)tar
	-test ! -f rmt || $(INSTALL) rmt /etc/rmt
	$(INSTALLDATA) $(srcdir)/tar.info* $(infodir)

$(OBJS): tar.h port.h testpad.h
regex.o buffer.o tar.o: regex.h
# getdate.y 有8個變換/減少衝突。

testpad.h: testpad
	./testpad

testpad: testpad.o
	$(CC) -o $@ testpad.o

TAGS:	$(SRCS)
	etags $(SRCS)

clean:
	rm -f *.o tar rmt testpad testpad.h core

distclean: clean
	rm -f TAGS Makefile config.status

realclean: distclean
	rm -f tar.info*

shar: $(SRCS) $(AUX)
	shar $(SRCS) $(AUX) | compress \
	  > tar-`sed -e '/version_string/!d' \
		-e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
		-e q
		version.c`.shar.Z

dist: $(SRCS) $(AUX)
	echo tar-`sed \
	     -e '/version_string/!d' \
	     -e 's/[^0-9.]*\([0-9.]*\).*/\1/' \
	     -e q
	     version.c` > .fname
	-rm -rf `cat .fname`
	mkdir `cat .fname`
	ln $(SRCS) $(AUX) `cat .fname`
```

```
	tar chZf `cat .fname`.tar.Z `cat .fname`
	-rm -rf `cat .fname` .fname

tar.zoo: $(SRCS) $(AUX)
	-rm -rf tmp.dir
	-mkdir tmp.dir
	-rm tar.zoo
	for X in $(SRCS) $(AUX) ; do \
	    echo $$X ; \
	    sed 's/$$/^M/' $$X \
	    > tmp.dir/$$X ; done
	cd tmp.dir ; zoo aM ../tar.zoo *
	-rm -rf tmp.dir
```

腳注

(1)

爲 MS-DOS 和 MS-Windows編譯的GNU Make和將前綴定義爲DJGPP樹體系的根的行爲相同。

(2)

在MS-DOS上, 當前工作目錄的值是global, 因此改變它將影響這些系統中隨後的命令行。

(3)

名詞翻譯對照表

archive	資料庫
archive member targets	資料庫成員目標
arguments of functions	函數參數
automatic variables	自動變數
backslash (\)	反斜線(\)
basename	基本檔案名
binary packages	二進制封包
compatibility	兼容性
data base	資料庫
default directries	預設目錄
default goal	預設最終目標
defining variables verbatim	定義多行變數
directive	指令
dummy pattern rule	僞樣式規則
echoing of commands	命令回顯
editor	編輯器
empty commands	空命令
empty targets	空目標

environment	環境
explicit rule	具體規則
file name functions	檔案名函數
file name suffix	檔案名後置
flags	標誌
flavors of variables	變數的特色
functions	函數
goal	最終目標
implicit rule	隱含規則
incompatibilities	不相容性
intermediate files	中間檔案
match-anything rule	萬用規則
options	選項
parallel execution	並行執行
pattern	樣式
pattern rule	樣式規則
phony targets	假想(phony)目標
prefix	前綴
prerequisite	先決條件
recompilation	重新編譯
rule	規則
shell command	shell命令
slash (/)	斜線(/)
static pattern rule	靜態樣式規則
stem	徑(stem)
sub-make	子make
subdirectories	子目錄
suffix	後置
suffix rule	後置規則
switches	開關
target	目標
value	值
variable	變數
wildcard	萬用字元
word	字